# NAVAL POSTGRADUATE SCHOOL
# Monterey, California

# THESIS

**HLA PERFORMANCE MEASUREMENT**

by

Ivan Chang Kok Ping

March 2000

| | |
|---|---|
| Thesis Advisor: | Michael Zyda |
| Thesis Co-Advisor: | Eric Bachmann |

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE**<br>March 2000 | **3. REPORT TYPE AND DATES COVERED**<br>Master's Thesis |

| | |
|---|---|
| **4. TITLE AND SUBTITLE**<br>HLA PERFORMANCE MEASUREMENT | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)**<br>IVAN CHANG KOK PING | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)** | **10. SPONSORING / MONITORING AGENCY REPORT NUMBER** |

**11. SUPPLEMENTARY NOTES**

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| | |
|---|---|
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT**<br>Approved for public release; distribution is unlimited. | **12b. DISTRIBUTION CODE** |

**13. ABSTRACT** ***(maximum 200 words)***

HLA uses an implicit Runtime Infrastructure (RTI) that completely encapsulates all simulation systems. This implementation on a networked virtual environment might be limited and could affect the overall system performance. The performance of HLA on PC workstations in a networked virtual environment might not be determined, and therefore the effects and limitations of its implementation could severely hamper the realism of real-time virtual environments. The goal of this thesis is to determine the limitations of the High Level Architecture (HLA) in a networked virtual environment on the Windows NT platform. In identifying the limitations of HLA, we will be able to ascertain the areas in which HLA can be improved. This thesis implements and measures the system performance of three different setups, namely a standalone virtual environment, a networked virtual environment using HLA, and a networked virtual environment using User Datagram Protocol (UDP). The system performance measured includes average CPU, network, graphics and memory processing requirements, frame rate per second, and the reliability of data received. The results indicate the use of heavily threaded processes by HLA significantly reduces overall system performance.

| | |
|---|---|
| **14. SUBJECT TERMS**<br>High Level Architecture, User Datagram Protocol | **15. NUMBER OF PAGES** |
| | **16. PRICE CODE** |

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT**<br>Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE**<br>Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT**<br>Unclassified | **20. LIMITATION OF ABSTRACT**<br>UL |

**NSN 7540-01-280-5500** **Standard Form 298 (Rev. 2-89)**
**Prescribed by ANSI Std. 239-18**

# HIGH LEVEL ARCHITECTURE PERFORMANCE MEASUREMENT

Ivan Chang Kok Ping
Singapore MINDEF
BEng, Loughborough University of Technology, 1993

Submitted in partial fulfillment of the
Requirements for the degree of

**MASTER OF SCIENCE MODELING VIRTUAL ENVIRONMENTS AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL**
**March 2000**

Author: ________________________________________
Ivan Chang Kok Ping

Approved by: ________________________________________
Michael Zyda, Thesis Advisor

________________________________________
Eric Bachmann, Associate Advisor

________________________________________
Rudy Darken, Academic Associate
Modeling Virtual Environments and Simulation
Academic Group

## ABSTRACT

HLA uses an implicit Runtime Infrastructure (RTI) that completely encapsulates all simulation systems. This implementation on a networked virtual environment might be limited and could affect the overall system performance. The performance of HLA on PC workstations in a networked virtual environment might not be determined, and therefore the effects and limitations of its implementation could severely hamper the realism of real-time virtual environments. The goal of this thesis is to determine the limitations of the High Level Architecture (HLA) in a networked virtual environment on the Windows NT platform. In identifying the limitations of HLA, we will be able to ascertain the areas in which HLA can be improved. This thesis implements and measures the system performance of three different setups, namely a standalone virtual environment, a networked virtual environment using HLA, and a networked virtual environment using User Datagram Protocol (UDP). The system performance measured includes average CPU, network, graphics and memory processing requirements, frame rate per second, and the reliability of data received. The results indicate the use of heavily threaded processes by HLA significantly reduces overall system performance.

# TABLE OF CONTENTS

# LIST OF FIGURES

# I. INTRODUCTION

## A. MOTIVATION

The High Level Architecture (HLA) is the Department of Defense standard for simulation interoperability that provides methods of defining how distributed simulations will communicate. HLA is fundamentally an architecture that facilitates the interoperability of distributed models and simulations. It utilizes Runtime Infrastructure (RTI) software to implement the interface specification and provides the network functions needed to accomplish data distribution. The RTI provides services that handle the interoperability of simulators across a network, it is responsible for ensuring that the interactivity within the real-time virtual environment is not compromised and that all other entities participating in the networked virtual environment are accurately represented. It is apparent that the RTI plays a key role in HLA implementation.

The repercussions of introducing a standard that might not meet real-time specifications would be devastating. Visual realism is dependent on the rate in which a graphics frame is presented. A rate of less than 10 frames per second would severely affect visual realism of a virtual environment. Due to the criticality of real-time presentation and the need to correctly represent participating entities, it is imperative that the RTI be designed to handle these issues. A good benchmark is to compare HLA performance against the User Datagram Protocol (UDP). UDP is one of the core data transmission protocols used for Distributed Interactive Simulation (DIS), a standard that was superseded by HLA.

This study analyzes the frame rate and reliability of the data transmitted via a comparison of these two methods and hence, establishes the limitations or improvements HLA has over the UDP form of data transmission.

## B. BACKGROUND

The Distributed Interactive Simulation (DIS) network software architecture evolved from Simulation Networking (SIMNET), a low-cost network virtual environment for training tank platoons. The reason for creating DIS was to allow any number of simulation sites to participate in a networked virtual environment on any type of platform and thereby create a much larger simulation arena at a lower cost. DIS was a huge success [Ref. 1]. However, the problem of discrepancies between entity position, orientation and line of sight computation complicated the interactions between entities. Furthermore, data representations of participating entities were left to each site to decide on how to model them. Although packet definition set of DIS was broad, it was not general enough to allow it to be implemented across all simulators. Packets containing entity state were large and mostly redundant. DIS also failed to adequately define ways to represent new types of information. These were normally packed into a Data PDU (Protocol Data Unit), which is meant to accommodate new undefined information.

In order to maintain consistent data representation, the Department of Defense (DoD) identified a need for a common high-level simulation architecture that would facilitate interoperability and software reuse. The goal was to minimize the cost of simulators and maximize the reuse of Modeling & Simulation (M&S) components. HLA was developed to fulfill this need [Ref. 1]. A team from the industry and government developed the initial HLA concept, and in the summer of 1995 three technical contractor teams and a program evaluation team were commissioned to conduct 6-month investigations of technical options and approaches. By March 1996, an initial technical architecture was formulated which defined the concept of a basic high level architecture. The baseline HLA technical approach was then developed based on specifications stipulated by Architecture Management Group (AMG). The AMG was made up of technical teams from 16 major defense programs that covered a wide range of defense simulation uses. The baseline was completed in August 1996 with the release of HLA 1.0. This spearheaded the implementation of the key supporting software architecture for end-user community.

This thesis describes an implementation that incorporates HLA to handle the network communication between each simulation site, and compares its performance against another implementation that uses UDP. The following sections provide an overview of HLA and UDP features and how they apply to this implementation. Later chapters provide a detailed description of both systems.

## C. HIGH LEVEL ARCHITECTURE

HLA is an architecture that allows different computer simulations to be combined into a large simulation arena, and therefore facilitates reuse of M&S software modules. This reduces the cost and time required to create a new virtual environment for training [Ref. 1]. HLA defines some terms that will be used in this thesis report. They are as follows:

- Federation. A combination of interacting simulation systems participating in the virtual environment arena.
- Federate. Each simulation participating in the federation. A federate could represent a tank simulator or the simulation of a battalion of tanks.
- Federation Execution (FedExec). A session of a federation in execution.

HLA will be covered in detail in Chapter II.

## D. USER DATAGRAM PROTOCOL

UDP is a connectionless data transport protocol that is unreliable. This service is unreliable because it neither guarantees delivery nor preserves the packet sequence. Any packet that is lost or late will not be reported. This often leads to data arriving out of order. However, UDP is simple and has a low overhead, which provides efficiency that can result in performance benefits [Ref. 3]. Since each entity updates its state frequently, a missing or late data packet would not cause any significant consequence to the overall simulation. Chapter III provides a detailed description of UDP.

## E. THESIS ORGANIZATION

This thesis is organized into the following chapters:

- Chapter I: Introduction. Discusses the motivation of conducting HLA performance measurements, and the events that led to the formation of HLA. This chapter also outlines the organization of the thesis.
- Chapter II: High Level Architecture. Discusses in detail the HLA architecture, and RTI implementation.

- Chapter III: User Datagram Protocol. Discusses in detail the UDP protocol.
- Chapter IV: Implementation. Describes the development of a networked virtual environment, and implementations of it using both HLA and UDP.
- Chapter V: Results and Limitations. Describes performance results of the implementation, and the limitations of HLA in comparison with UDP.
- Chapter VI: Conclusion and Recommendations. Discusses the significance of the results and recommends ways to improve HLA performance. Gives ideas as to future work that should be completed in this area.

THIS PAGE INTENTIONALLY LEFT BLANK

# II. HIGH LEVEL ARCHITECTURE (HLA)

## A. INTRODUCTION

HLA defines a software architecture. It is not a software implementation. HLA establishes a common high-level simulation architecture to facilitate the interoperability of all types of simulations, models, and C4I systems. HLA is designed to achieve standardization in the M&S community and to facilitate the reuse of M&S components.

HLA architecture implements an object-oriented network design. Each simulation system that is an interacting component of the HLA architecture is treated as an object and it is known as a federate. A collection of federates that participate in the same virtual environment is known as a federation. The federate registers its data members with a software, Run-Time Infrastructure (RTI), which defines the HLA architecture. This software coordinates and controls the data transfer between federates. All federates send and receive their data through the RTI. The RTI is responsible for handling low-level networking services, and for ensuring that the data are promptly sent from publishing federates to subscribing federates.

The HLA is defined by three components:

1. <u>Object Model Template (OMT)</u>. Provides a common method for recording information, and establishes the format of Federation Object Model (FOM).
2. <u>HLA Rules</u>. These ensure proper interaction of simulations in a federation, and describe the simulation and federate responsibilities.

3. Interface Specification. This identifies the callback functions each federate must provide, and defines the RTI services. The RTI is the implementation of HLA that provides network and simulation management services.

The following sections will cover the HLA components in detail.

## B. HLA COMPONENTS

### 1. Object Model Template (OMT)

All objects and interactions are defined according the standard OMT. The OMT provides a common framework for HLA object model information, and it promotes interoperability and reuse of simulations and its components. The FOM describes the objects and interactions that are shared with other federate. It does not describe things internal to a single federate. The two main classes of OMT are the following:

- Interaction classes.
- Object classes.

#### *1.1 Interaction Classes*

Interaction classes are comprised of parameters. They represent an occurrence or specific event in the simulation. These parameters are sent once through the RTI to other federates. The parameters do not persist after they have been received.

#### *1.2 Object Classes*

Object classes are comprised of attributes. An example of an object is a "tank" which has attributes such as size, weight, and range, and they persist or have continued existence in the simulation. Federates update the state of an object instance by providing new values for its attributes.

### *1.3 Interactions and Objects Classes*

A federation can be described completely in term of interactions, objects or both. Guidelines for selecting class are as follows:

- An event or occurrence should be represented as an interaction.
- An entity that has persistent state should be represented as an object.

## **2. HLA Rules**

The HLA rules describe the responsibilities of federates and their relationships with the RTI. The first five rules deal with federations, and the latter five with federates.

### *2.1 Federation Rules*

- Rule 1. Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
- Rule 2. In a federation, all representations of objects in the FOM shall be in the federate, not in the run-time infrastructure (RTI).
- Rule 3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
- Rule 4. During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.

- Rule 5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

### *2.2 Federate Rules*

- Rule 6. Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
- Rule 7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
- Rule 8. Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOM.
- Rule 9. Federates shall be able to vary the conditions (e.g. thresholds) under which they provide updates of attributes of objects, as specified in their SOM.
- Rule 10. Federates shall be able to manage local time in a way which will allow them to coordinate data exchange with other members of a federation.

## 3. Interface Specification

The HLA interface specification defines the data exchanges that take place between each federate and the federation. There are six management areas in the FedExec life cycle, each defines the services that handle specific tasks to be executed to manage the federation.  Federates do not communicate with each other directly instead they invoke services through the RTI.  These services are either RTI-initiated or federate-initiated services, and they reside in either the RTI or federate respectively.  The RTI presents an interface called the RTIambassador to each federate, and likewise each federate offers an interface called the FederateAmbassador to the RTI.  The RTIambassador does not differentiate between federates and therefore it has a common interface.  However each federate uses those RTI services appropriate for its purpose and its FederateAmbassador may be unique.  Each federate has a single point of contact with the RTI regardless of the number of processes or computers needed for the federate to execute its simulation.

The six management areas are summarized as follows:

- Federation Management.  This area defines tasks which manage a federation execution.  It includes tasks such as creating federations, joining federates to federations, observing federation-wide synchronization points, saving and restoring federation states, resigning federates from federations, and destroying federations.
- Time Management.  This task management area controls the ordering of events in logical time advancement.  The logical time does not represent any unit of real time.  This service allows each federate to advance its logical time in coordination with other federates.  It controls the delivery of time-stamped events so that each federate gets updated events.  A federate can be either time-constrained or time-regulating or both or

neither. In the time-constrained mode, the federate advance of logical time is constrained by other federates, and in the time-regulating mode the federate advance of logical time regulates other federates. The choice of time management is left to the federate's mode of operation.

- Declaration Management. This specifies data a federate will send and receive, and controls where the data is sent based on other federate interests. Its tasks include publishing objects or interactions that a federate intents to produce, subscription to specific data, and controlling the data flow by informing a federate whether other federates have subscribed to the data it intends to produce, so that it can stop producing the information when it is not needed.
- Object Management. This service is used to send and receive interactions, and register new instances of an object class and update its attributes. Its tasks include creating, modifying, and deleting objects and interactions, managing object identification, facilitating object registration and distribution, coordinating attribute updates among federates, and accommodating various transportation and time management schemes.
- Ownership Management. This service supports the sharing or transfer of ownership for individual object attributes, and it offers both push and pull based transactions. The RTI allows federates to share the responsibility for updating and deleting object instances, however only one federate can update an attribute of an object at any given time. If the object is

completely owned by one federate, then that federate is responsible for updating the attributes. An object instance may have various attributes owned by different federates however an attribute can only be owned by at most one federate. Moreover, only one federate has the privilege to delete the object instance.

- Data Distribution Management. This service supports efficient routing of data among federates, and it provides a flexible and extensive mechanism for isolating publication and subscription to regions of interest.

These groups of services are designed to be independent and they can be used without referencing each other. The default services provided by the ownership, time, and data distribution management groups are adequate, however the federation, declaration, and object management groups need to be defined according to the needs of each federate.

## C. RUN-TIME INFRASTRUCTURE (RTI) IMPLEMENTATION

This section describes the RTI overview, and the primary RTI implementation of RTIambassador and FederateAmbassador functions that support the six management areas mentioned in the previous section.

### 1. RTI Overview

RTI is the software that implements the HLA interface specification. It provides an architectural foundation of common services to simulation systems and thereby promotes portability and interoperability. These services include construction and destruction of federations, support of object declaration and management between federates, providing communications between federates, and managing federation time. The main RTI components are described in the following sub-section.

### *1.1 RtiExec – The RTI Executive*

The RtiExec is a global process. It executes on one platform and listens to a well-known port. Each federate communicates with the RtiExec to initialize their RTI components. The role of the RtiExec is to manage the creation and destruction of FedExecs, direct new participating federates to the appropriate Federation execution, and ensure that each FedExec has a unique name.

### *1.2 FedExec – The Federation Executive*

Each executing federation has one FedExec process. Its role is to manage a federation that comprises multiple federates. It allow federates to join and resign, and facilitates data exchange between participating federates. The FedExec is created by the first federate that successfully invokes the Create Federation Executive service of a particular federation execution. Every federate of the federation is then assigned a unique handle.

### *1.3 librti – The RTI Library*

The RTI library extends the services specified in the HLA interface specification to the federate. Federates use the librti to implement methods to communicate with the rtiexec, fedexec and other federates. All requests made by a federate on the RTI take the form of a RTIambassador method call. The FederateAmbassador identifies the callback functions each federate is required to provide.

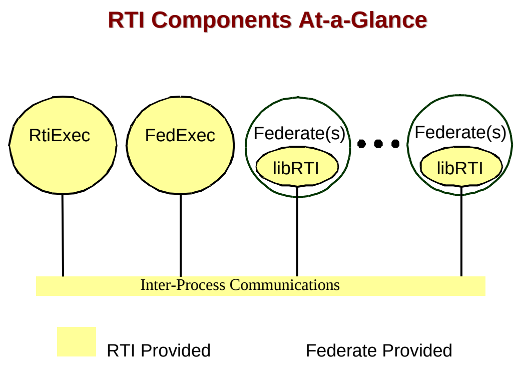

Figure 1. RTI Components At-a-Glance. [From Ref. 4].

## 2. Federation Management

The key functions for federation management reside mainly in the RTIambassador. The following diagram illustrates its life cycle.

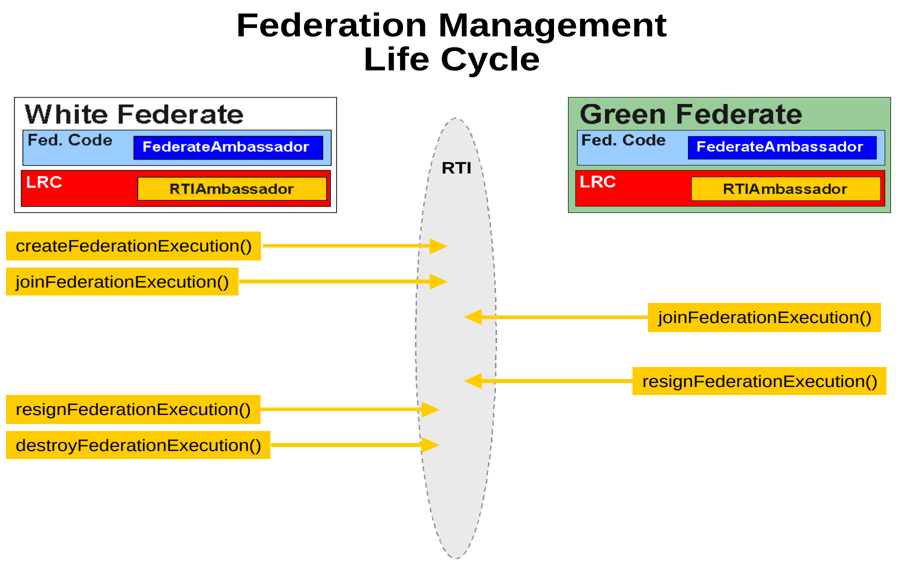
Federation Management
Life Cycle
White Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
Green Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
RTI
createFederationExecution()
joinFederationExecution()
joinFederationExecution()
resignFederationExecution()
resignFederationExecution()
destroyFederationExecution()

Figure 2. Federation Management Life Cycle. [From Ref. 4].

### *2.1 createFederationExecution()*

This function initiates the Local RTI Component (LRC) communication with the rtiexec process. If the specified federation does not exist, the rtiexec process will create a new FedExec process and link the federation to it. Otherwise, it will return a FederationExecutionAlreadyExists exception, which can be ignored. This allows the software function to be re-used, as such calls by participating federates would create the federation if none already exists. Normally the first federate will attempt to create the FedExec.

### *2.2 joinFederationExecution()*

This function associates a federate with an existing federation execution. If the federation execution does not exist when the call to join was made, the FedExec may not be able to initiate communication with the federate that successfully created the FedExec.

### *2.3 tick()*

This function is not specified in the Federation Management. Nevertheless, it plays a very important role for the LRC. The tick function yields time for the LRC to perform its task of exchanging information with other federates. It does not advance time as a result of its operation. If insufficient time is allocated to tick the LRC, major federation problems may occur. For instance, a late joining federate to the federation would be blocked if the existing federates did not perform ticks on their LRC. There are two methods to tick the LRC. One has no arguments and the other has two arguments. The first method yields time to the LRC to complete each major task, but there is no guarantee as to the amount of time required to complete its task. The second method is similar to the first, except that it specifies the lower and upper bounds of time being allocated to the tick function.

### *2.4 resignFederationExecution()*

This function removes a participating federate from the federation. Since the federate may be responsible for updating an object, the federate can then take any of the following appropriate action on the objects upon resignation of a federate: ‘release attributes’, ‘delete objects’, ‘delete objects and release attributes’, or take ‘no action’.

### *2.5 destroyFederationExecution()*

This function terminates an executing federation. If the federate is not the last participating federate to terminate, it will throw a FederatesCurrentlyJoined exception.

### *2.6 Federate Synchronization*

The synchronization between federates is handled by the RTI, since it is able to exchange timing information and coordinate activities between federates. The Federation Management allows federates to communicate explicit synchronization points. The following diagram illustrates the synchronization activities.

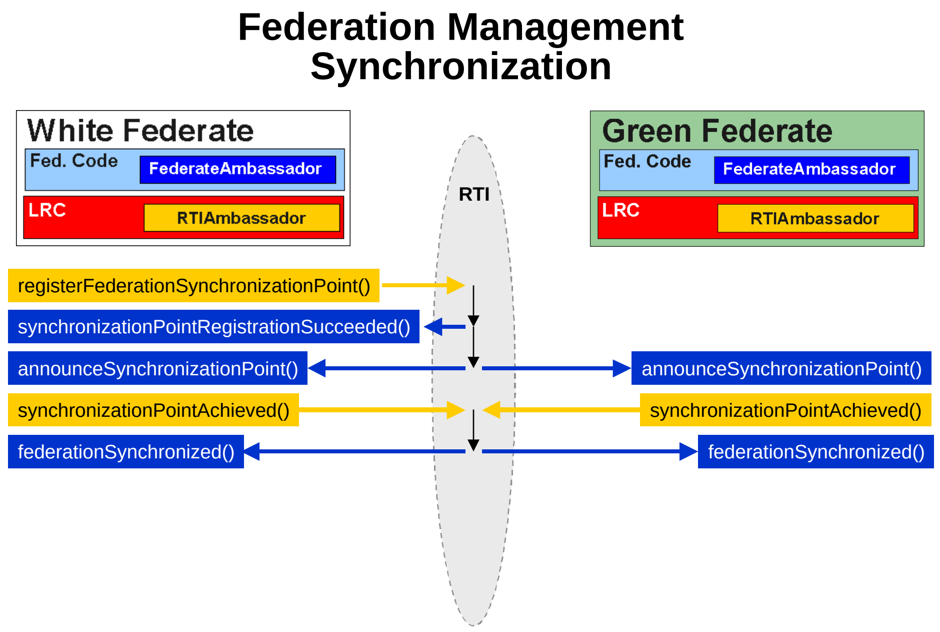
Federation Management
Synchronization
White Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
Green Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
RTI
registerFederationSynchronizationPoint()
synchronizationPointRegistrationSucceeded()
announceSynchronizationPoint()
announceSynchronizationPoint()
synchronizationPointAchieved()
synchronizationPointAchieved()
federationSynchronized()
federationSynchronized()

Figure 3. Federate Management Synchronization. [From Ref. 4].

### *2.7 Save and Restore*

The RTI provides functions that support federation-wide saves and restores. The following figures illustrates these activities.

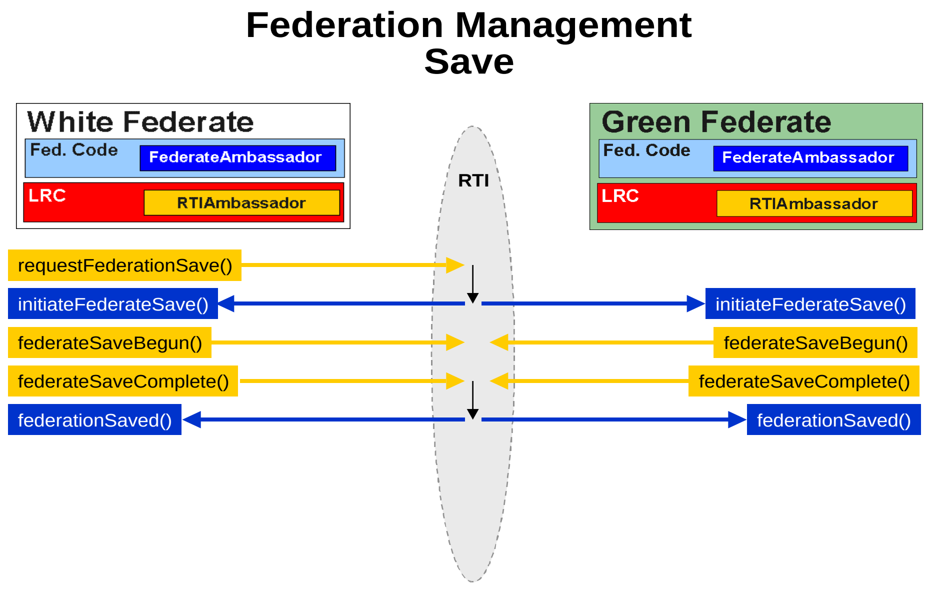

Figure 4. Federation Management Save. [From Ref. 4].

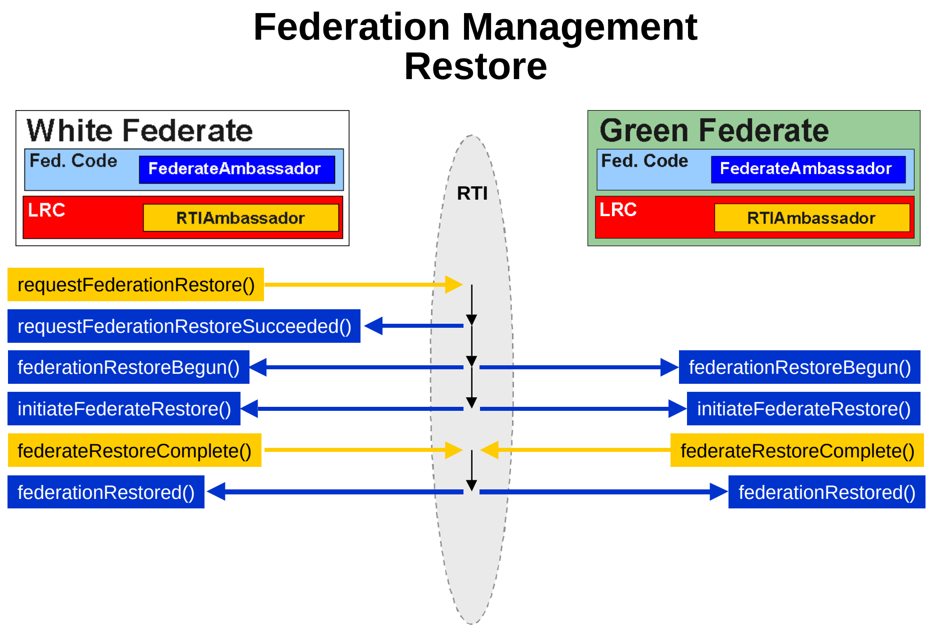

Figure 5. Federation Management Restore. [From Ref. 4].

## 3. Time Management

This section covers the mechanics of the RTIambassador service and FederateAmbassador callback methods, which support time management functionality.

### *3.1 Regulating and Constrained Status*

The diagram identifies the RTIambassador and FederateAmbassador functions associated with enabling and disabling the regulating or constrained status of a federate.

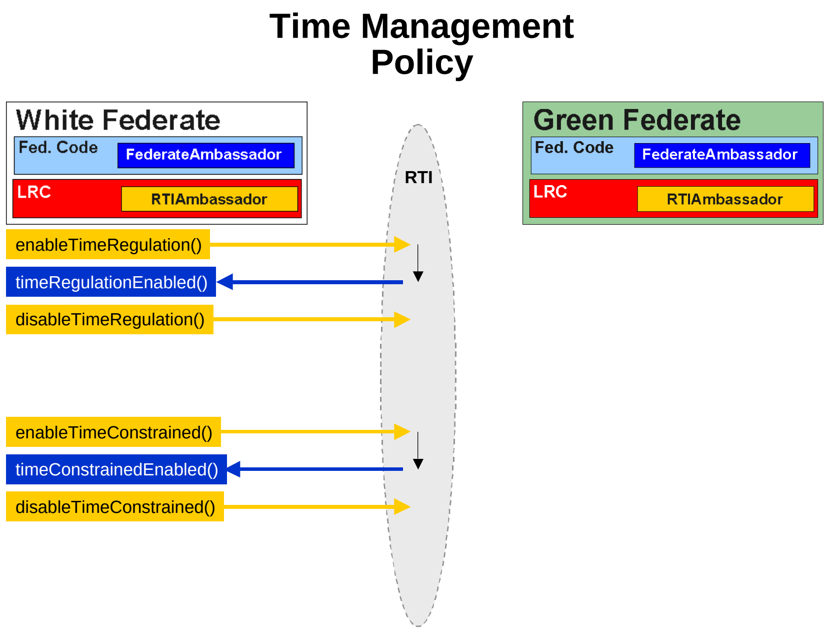

Figure 6. Toggling Regulating and Constrained Status. [From Ref. 4].

3.1.1 Regulation Policy. By default, the regulation status of a federate is disabled. The federate enables this status by making a request to the RTI using the RTIambassador member function enableTimeRegulation(). The local RTI component (LRC) calls the FederateAmbassador callback function timeRegulationEnable() to inform the federate that its request has been granted and inform the federate of its new logical time. The federate can disable the time regulation at anytime by calling the RTIambassador function disableTimeRegulation().

3.1.2 Constrained Policy. A federate has the constrained status disabled by default. This status is enabled by the federate using the RTIambassador member function enableTimeConstrained() to make its request to the RTI. The LRC calls the FederateAmbassador callback function timeConstrainedEnable() to inform the federate that its request has been granted. The time constrained status can be disabled at anytime by the federate through the use of the RTIambassador function disableTimeConstrained().

### *3.2 Time Advance Requests*

There are three types of time advance requests that a federate can make during execution. The time advance services are time-step, event-based, and optimistic federates.

3.2.1 Time-Stepped Federates. This federate will perform computation on its values at the current time and all events that will occur before the next time step. When a timeAdvanceRequest() is made by the federate, the federate continues to process its values until the LRC has released all time-stamp ordered messages that have a time-stamp less than or equal to the next time-step from its FIFO queue. Once the LRC has sent all the orders it will call the FederateAmbassador callback function timeAdvanceGrant(). The federate time then increases by a time-step.

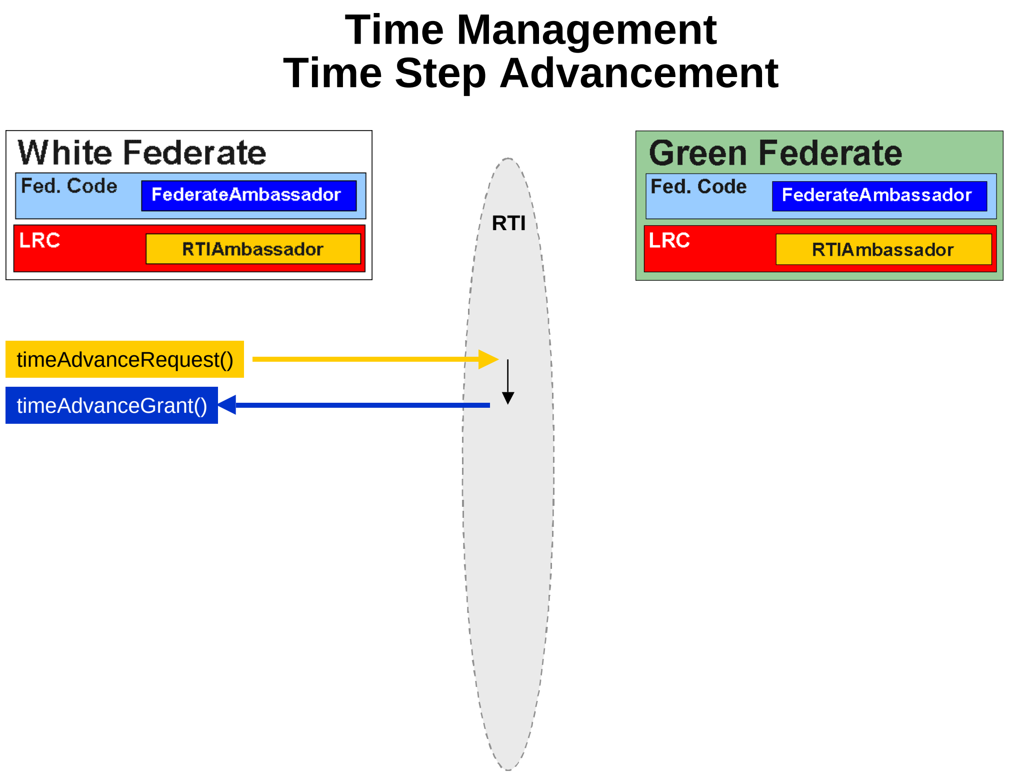

Figure 7. Time Step Advancement. [From Ref. 4].

3.2.2 Event-Based Federates. This federate will increment its time based on the time of the received event. When a nextEventRequest() is made by the federate, the LRC will release any order messages from its FIFO queue and all time-stamp ordered messages that have a time-stamp equal to the minimum next event time. Once all messages with a time equal to the minimum next event time have been sent, the LRC will initiate the callback function timeAdvanceGrant() via the FederateAmbassador. The federate time will then increase to the next minimum event time.

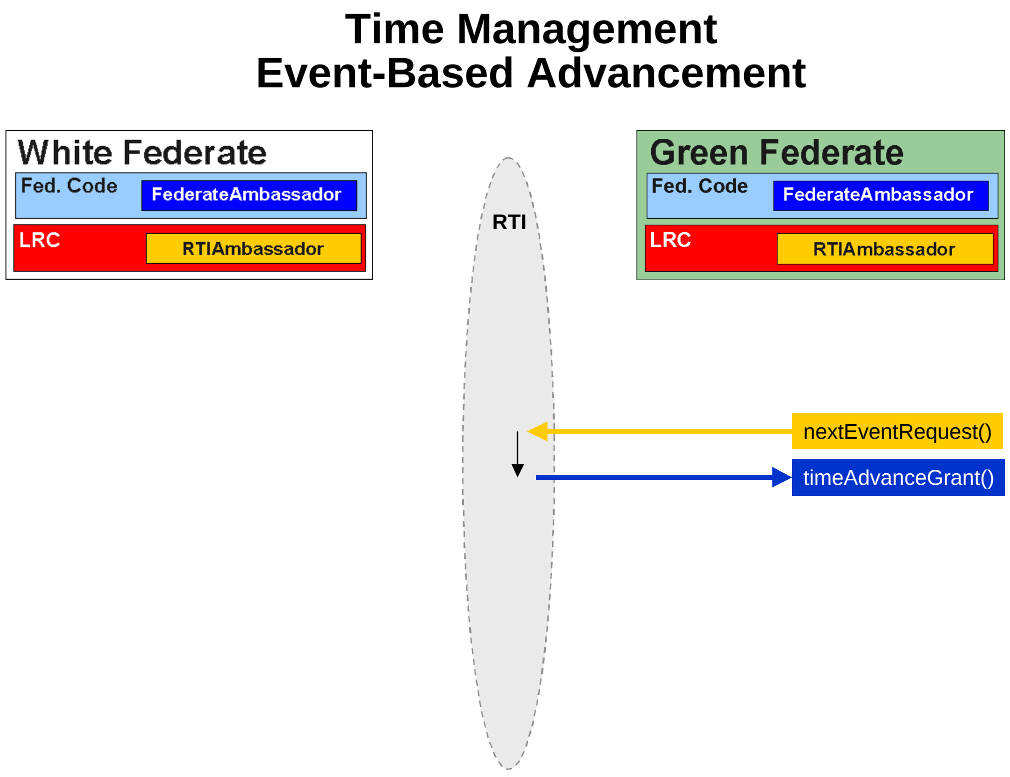

Figure 8. Event-Based Advancement. [From Ref. 4].

3.2.3 Optimistic Federates. These federates are not constrained by the time advancement of regulating federates, but instead will proceed to compute and send events in the future, and to receive any events sent to the federation execution regardless of time-stamp ordering. When a flushQueueRequest() is made by the federate, the LRC will release all receive order and time-stamp ordered messages from its FIFO queue. Once all messages have been sent, the LRC will initiate a timeAdvanceGrant() callback function via the FederateAmbassador.

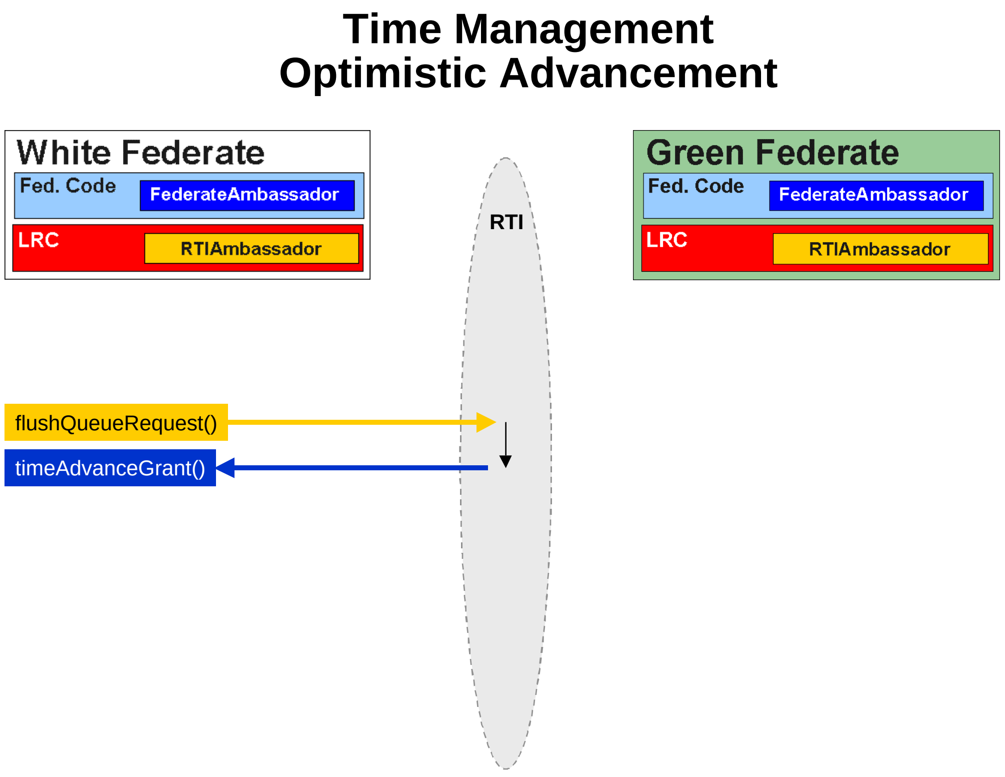

Figure 9. Optimistic Advancement. [From Ref. 4].

## 4. Declaration Management

Declaration management includes publication, subscription and associated control functions. This management area declares the objects and/or interactions that a federate is able to publish, and as well as the federate’s subscription interest in objects and/or interactions. The RTI monitors the data produced and consumed by federates, and thereby controls the data flow between federates.

### *4.1 Object Publication and Subscription*

The federate declares its publication and subscription interests to the LRC by calling the functions subscribeObjectClassAttributes() and publishObjectClass() of the RTIambassador. An AttributeHandleSet identifies a set of attributes. A federate can declare an interest to publish or subscribe to an object class by executing the following steps:

- Obtain a handle for the current object class.
- Use the static create() method in the class AttributeHandle-SetFactory to create a free-store allocated AttributeHandleSet.
- For each attribute the federate can publish:
  - a. Obtain the handle for the current attribute.
  - b. Add the handle to the AttributeHandleSet.
- Publish and/or subscribe to the AttributeHandleSet for the object class.

Whenever there is a call to publishObjectClass() and subscribeObject-Class() for an object, it will supersede all earlier calls. Once a federate is no longer interested in any attributes of an object class, it should call the methods unpublishObjectClass() and unsubscribeObjectClass().

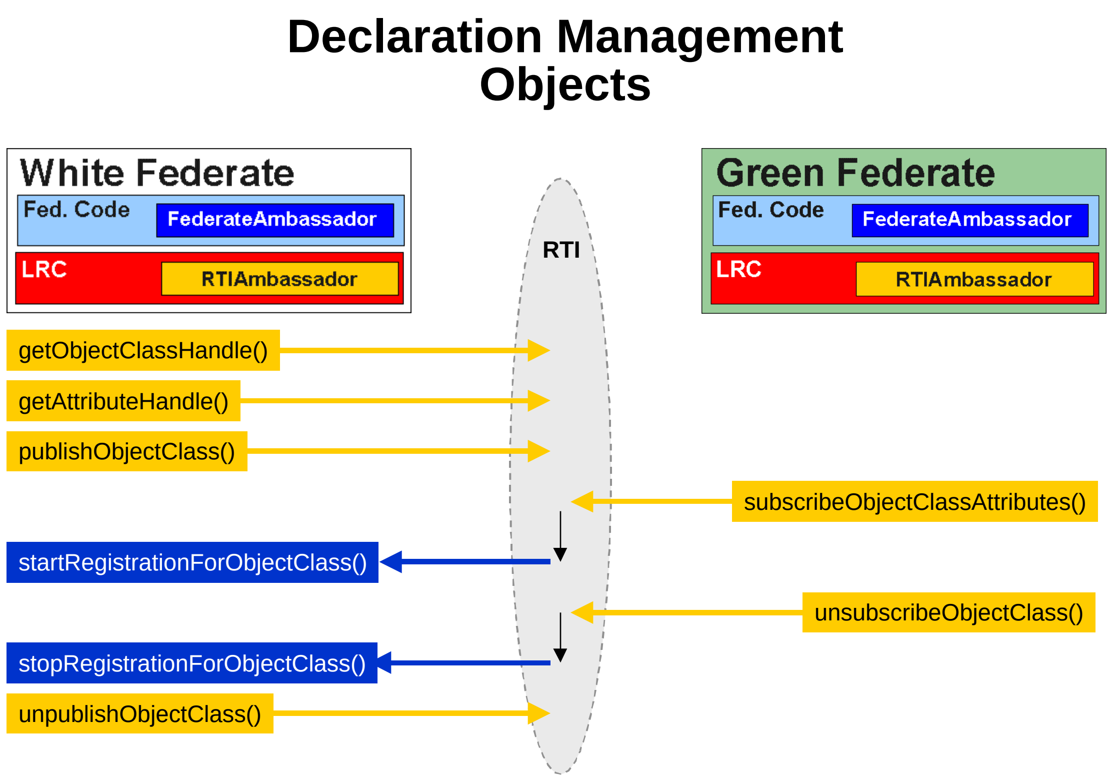

Figure 10. Object Publication and Subscription. [From Ref. 4].

### *4.2 Interactions Publishing and Subscribing*

The federate cannot specify interest in particular interaction parameters. It has to publish and/or subscribe to either all or none of the parameters. The interaction interest can be declared dynamically. Every call to publishInteractionClass() and subsribeInteractionClass() for an interaction class supersede all earlier calls. Whenever a federate is no longer interested in an interaction class it should call the methods unpublishInteractionClass() and unsubscribeInteractionClass().

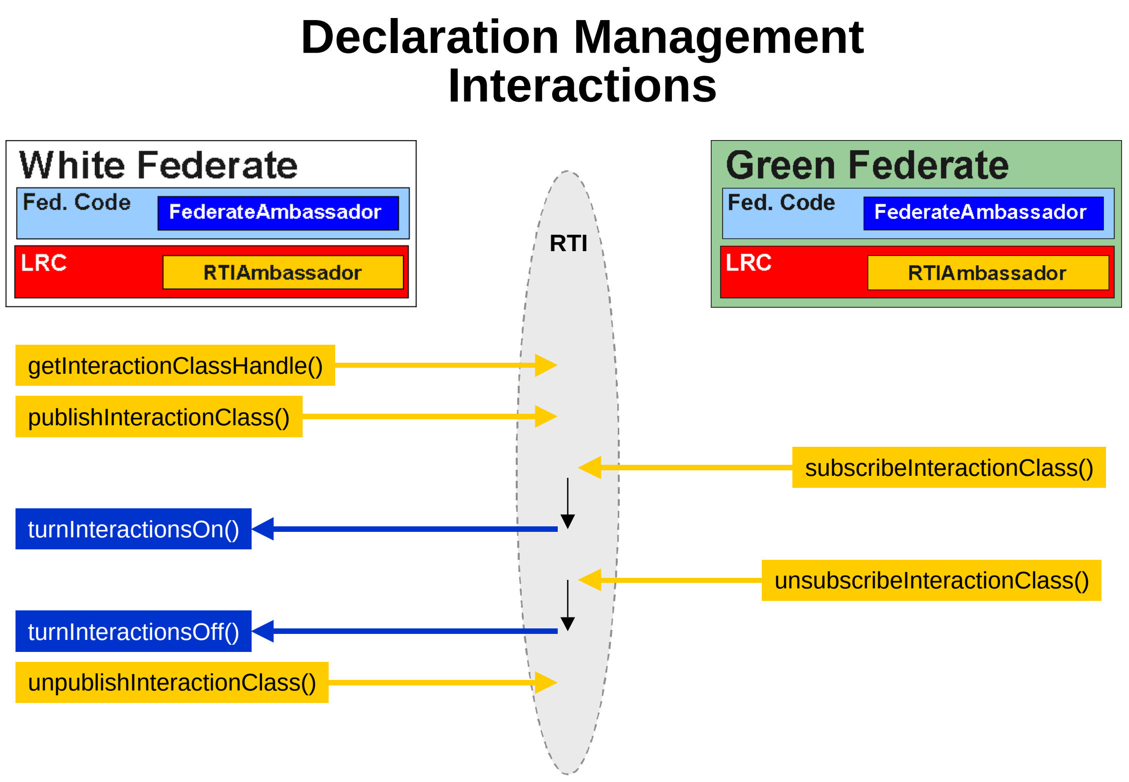
Declaration Management Interactions
White Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
Green Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
RTI
getInteractionClassHandle()
publishInteractionClass()
subscribeInteractionClass()
turnInteractionsOn()
unsubscribeInteractionClass()
turnInteractionsOff()
unpublishInteractionClass()

Figure 11. Interaction Publication and Subscription. [From Ref. 4].

## 5. Object Management

This management service includes instance registration and updates on the object production side, and instance discovery and reflection on the object consumer side. This service also includes sending and receiving interactions, controlling instance updates based on consumer demand, and other support functions.

### *5.1 Registering, Discovering, and Deleting Object Instances*

The RTIambassador method registerObjectInstance() informs the LRC that a new object instance has joined the federation. This method returns an RTI::ObjectHandle that the LRC uses to identify the object instance, however it does not provide attribute values for the instance. Discovery is the counterpart to registration.

The FederateAmbassador callback function discoverObjectInstance() informs the participating federate that a new object instance has come into existence. This method returns an ObjectHandle that will be used to identify the object for subsequent updates. The LRC also initiates the callback function turnUpdatesOnForObjectInstance() to inform a federate whether there is an external interest in updates for the specific attributes of object instance. The federate that creates the object has the privilege of deleting it, however this responsibility can be handed to other federates. Nevertheless, only one federate can deleted each object. The RTIambassador method deleteObjectInstance() removes a registered object, and the LRC callback function removeObjectInstance() informs other participating federates that a previously discovered object is not available.

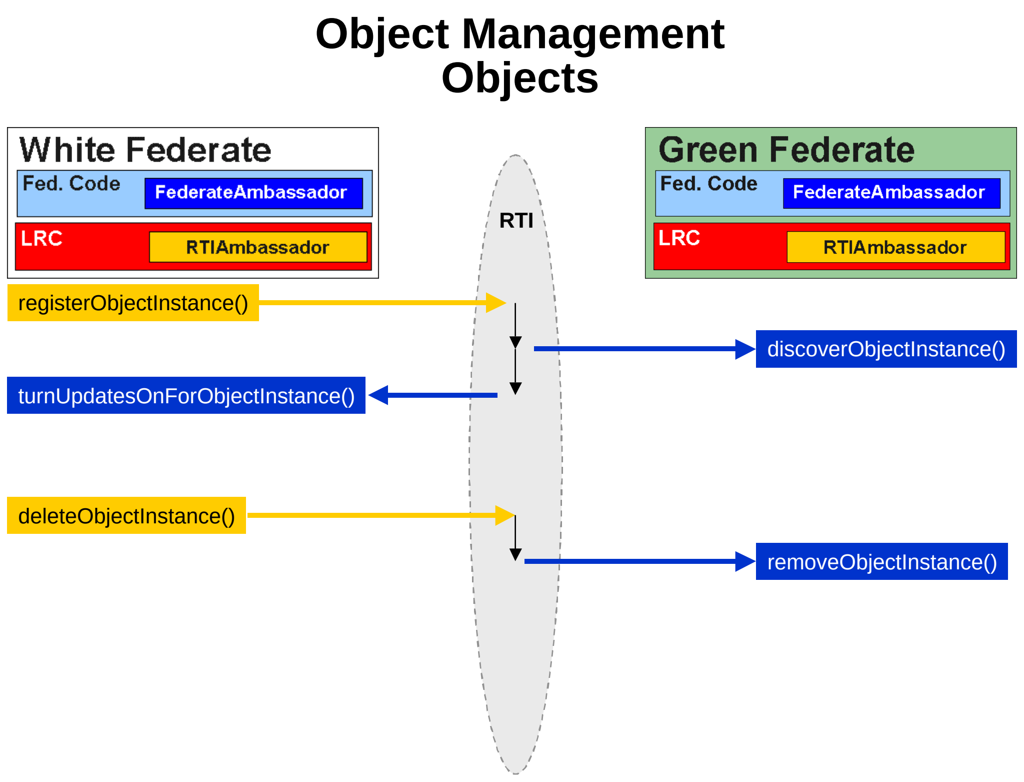

Figure 12. Registering, Discovering, and Deleting Object Instances. [From Ref 4].

### *5.2 Updating and Reflecting Object Attributes*

The federate must set up a RTI::AttributeHandleValuePairSet to update object attributes. An AttributeHandleValuePairSet identifies a set of attributes and their values. The static create() method in the class AttributeSetFactory is then used to create a free-store allocated AttributeHandleValuePairSet instance. The RTIambassador method updateAttributeValues() provides a means to update an attribute of an object instance. This method requires three arguments, an ObjectHandle which is provided by discoverObjectInstance(), an AttributeHandleValuePairSet, and a descriptive character string (tag – normally a NULL value). The fourth optional argument is time, this is used when the federate is regulating or an attribute is time-stamp ordered. Reflection is the counterpart to attribute updates. The LRC initiates the FederateAmbassador callback method reflectAttributeValues() to update the attributes of other participating federates whenever a federate calls updateAttributeValues().

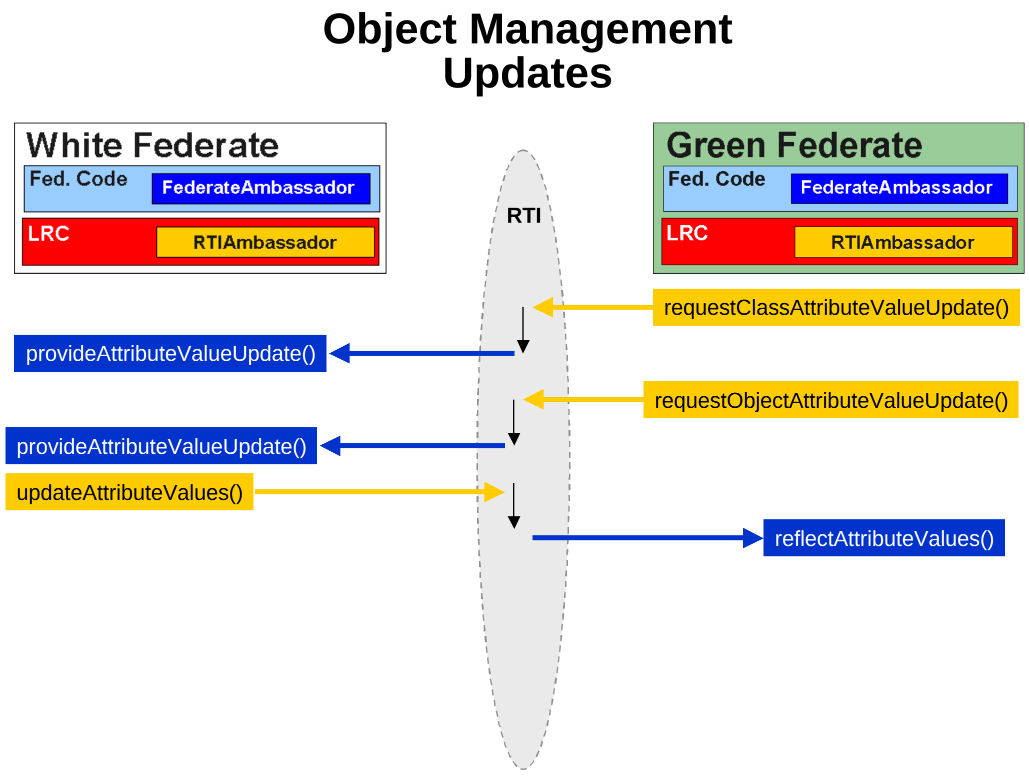

Figure 13. Updating and Reflecting Object Attributes. [From Ref. 4].

### 5.3 Encoding and Object Update

Whenever a federate sends an Object, it is responsible for encoding the data. The LRC does not know of the data content, as it only needs to know the object class name, attributes names, and the handle representation of the object and its attributes.

### 5.4 Decoding and Object Reflection

The receiving federate is responsible for decoding the data in the same order which the data was initially encoded. The FederateAmbassador callback method reflectAttributeValues() provides the AttributeHandleValuePairSet from which the federate can extract its data.

### *5.5 Exchanging Interactions*

Each interaction is constructed, sent and forgotten. The federate receives the interaction, decodes it and applies it in its simulation.

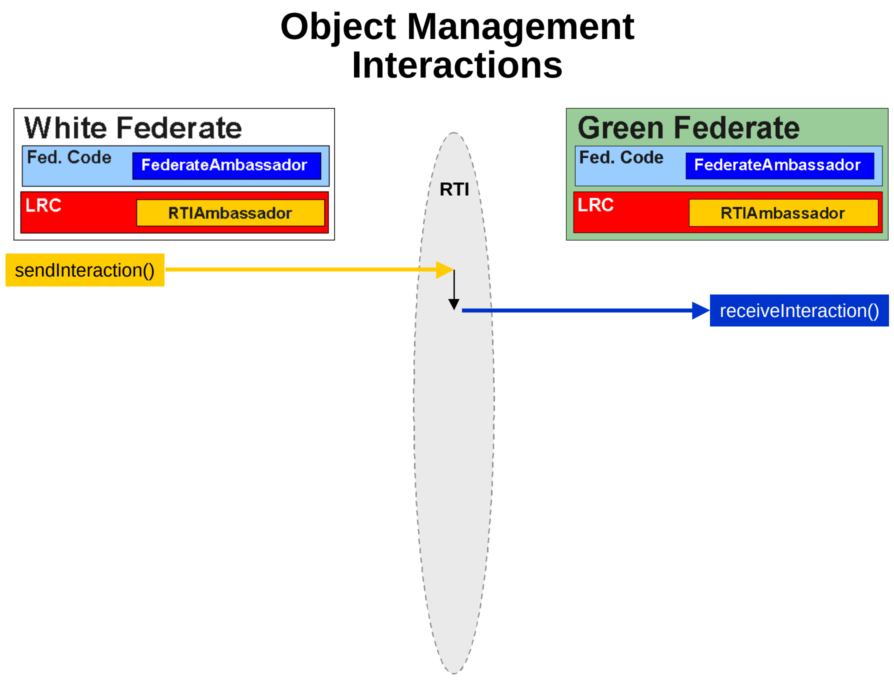

Figure 14. Exchanging Interactions. [From Ref. 4].

### *5.6 Object Control*

The object attributes update and interaction can be sent using one of two data transportation schemes, “reliable” or “best effort”. This transportation scheme is specified at the individual attribute level and interaction level for objects and interactions respectively. This declaration is described in the Federation Execution Data (FED) file. The transportation scheme of attributes and interaction can be changed dynamically using the RTIambassador method changeAttributeTransportType() and changeInteractionTransportType() respectively.

5.6.1 Attribute Management. The turnUpdatesOnForObject-Instance() callback is issued by the LRC when at least one participating federate is interested in updates for a particular object instance. If there's no further interest in the object instance, the LRC will call turnUpdatesOffForObjectInstance() to inform the federate to stop providing updates.

5.6.2 Enable and Disable Attribute Management. The RTIambassador provides methods to enable or disable the attribute management callbacks (described in 4.6.1) by invoking enableAttributeRelevanceAdvisorySwitch() and disableAttributeRelevanceAdvisory-Switch().

## 6. Ownership Management

This management service includes methods for registering and updating object instances. The RTI allows federates to be solely responsible or share the responsibility for updating and deleting object instances with a few restrictions. At any given time, only one federate has the responsibility of updating an attribute of an object instance and the privilege of deleting an object instance.

### *6.1 Push and Pull*

The exchange of attribute ownership can be pushed and/or pulled between federates. A federate that gives away the ownership of an attribute uses the push model, however it cannot push this responsibility to any federate that does not want ownership. Similarly, a federate that tries to obtain the ownership of an attribute uses the pull model, and it cannot pull this responsibility from any federate that does not want to release its responsibility.

### *6.2 Privilege to Delete*

The federate that owns the attribute has the privilege of deleting it. Likewise, this privilege to delete attribute can be exchanged between federates.

### *6.3 Ownership Pull*

The requesting federate must create an attribute handle set before it can request ownership of attributes from an object instance. The federate can invoke either of two RTIambassador methods while attempting to takeover the ownership of an attribute. The method attributeOwnershipAcqusition() tries to secure ownership of an attribute whether or not it is currently owned by another federate. The method attributeOwnershipAcquisitionIfAvailable() tries to secure ownership of attributes that are not owned by another federate. The method attributeOwnershipAcquisition() invokes requestAttributeOwnershipRelease() callback if the requested attributes are owned by other federates. Upon receiving this callback, a federate would respond with the RTIambassador method attributeOwnershipReleaseResponse(). If it could release the ownership of the attribute, it would respond with a null attribute handle set to indicate that the ownership cannot be released. When attributeOwnershipAcquisitionIfAvailable() is called, any attributes that are already owned would invoke attributeOwnershipUnavailable() callback.

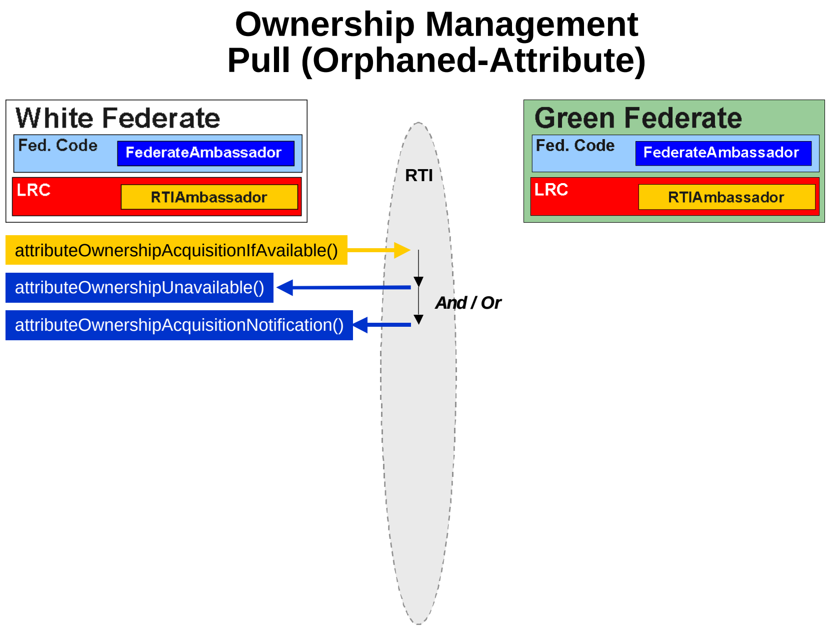

Figure 15. Pulling Orphaned Attribute. [From Ref. 4].

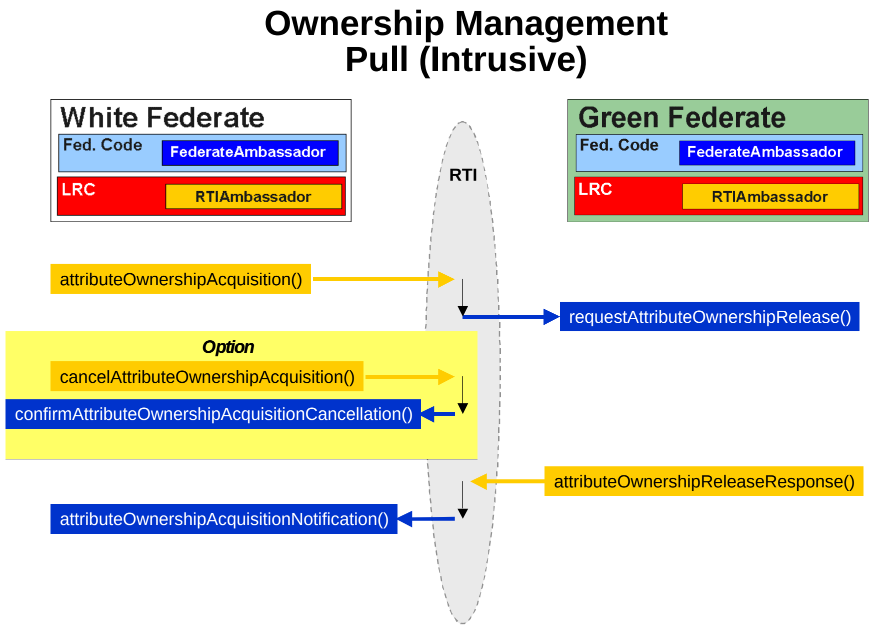
Ownership Management
Pull (Intrusive)
White Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
RTI
Green Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
attributeOwnershipAcquisition()
requestAttributeOwnershipRelease()
Option
cancelAttributeOwnershipAcquisition()
confirmAttributeOwnershipAcquisitionCancellation()
attributeOwnershipReleaseResponse()
attributeOwnershipAcquisitionNotification()

Figure 16. Pulling Ownership From Another Federate. [From Ref. 4].

## *6.4 Ownership Push*

The federate may release ownership of an attribute either 'unconditionally' or by 'negotiation'. An unconditional push will release a federate from attribute update and/or delete responsibilities without any commitment from another federate to assume these responsibilities. The federate can call the RTIambassador method unconditionalAttributeOwnershipDivestiture() to release its ownership responsibility immediately. Negotiated push requires that a federate retain responsibility until a new owner is found. The federate that initiates the negotiated push calls the RTIambassador method negotiatedAttributeOwnershipDivestiture(). The other participating federates that are capable of publishing those attributes are notified of the ownership push via the FederateAmbassador callback function requestAttributeOwnershipAssumption(). A federate that wished to obtain ownership of the attributes would invoke either the attributeOwnershipAcquisition() or attributeOwnershipAcquisitionIfAvailable() method. Once a new owner is found, the federate that initiated the push will receive a callback attributeOwnershipDivestitureNotification() to notify the federate that it is no longer responsible for the attribute. The new owner will receive the callback method attributeOwnershipAcquisitionNotification().

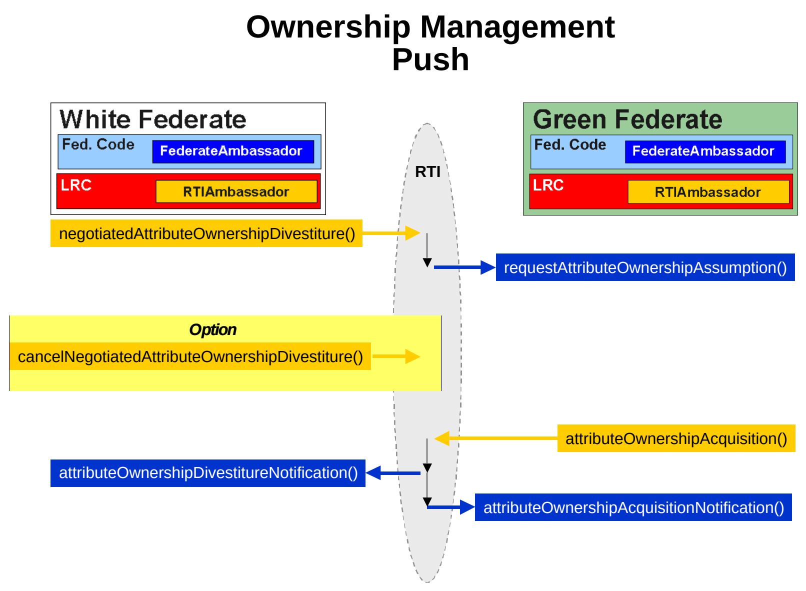
Ownership Management
Push
White Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
Green Federate
Fed. Code
FederateAmbassador
LRC
RTIAmbassador
RTI
negotiatedAttributeOwnershipDivestiture()
requestAttributeOwnershipAssumption()
Option
cancelNegotiatedAttributeOwnershipDivestiture()
attributeOwnershipAcquisition()
attributeOwnershipDivestitureNotification()
attributeOwnershipAcquisitionNotification()

Figure 17. Pushing Ownership To Other Federates. [From Ref. 4].

### 7. Data Distribution Management (DDM)

This management service is entirely optional and the federate need not use this service at all. DDM defines the routing spaces and regions that do not require the RTI to have knowledge about a Federation's data. This is defined by associating data with regions. A routing space defines the problem space, and it identifies all the dimensions on which a region might be defined. All federates that use a routing space must agree upon the dimensions of the routing space as well as the worst case upper and lower bounds along each dimension. The FED file specifies the routing spaces and dimensions available to the federate.

# III. USER DATAGRAM PROTOCOL (UDP)

## A. INTRODUCTION

UDP is a communication method often known as connectionless transport. It is connectionless because it does not need to establish point-to-point connection between systems across the network. The connectionless transport service is simple, and it offers limited services when data is exchanged between systems in a network that use the Internet Protocol. This limited service may cause UDP to be unreliable because it neither guarantees delivery nor preserves the packet sequence. This implies that data may be lost or late during transmission, therefore they would not be received in the order that they were sent. However, its simplicity has advantages over other transport services. It does not need to detect data integrity, establish an explicit connection, and maintain the connection. This makes it easy to implement and the low overhead, which provides efficiency, can result in better performance benefits. Therefore, data can be broadcasted or multicasted to many systems at once. UDP is appropriate for large-scale distributed simulation systems where each system transmits data to many participating systems. This is ideal for updating data parameters between simulation systems, since any transmitted data lost would be quickly replaced by the next update received, therefore reliability is not as important as timeliness of the data exchanged. UDP is one of core data transmission protocols used by DIS.

## B. ESTABLISHING CONTACT

A socket is a software point of contact between systems on the network. Before any communication between systems on the network can be established, the systems' sockets have to support UDP. A socket can be obtained by calling the socket() function which takes in three arguments; socket domain, socket type, and name of protocol. The socket domain used for internet communication is PF_INET, and the socket type for UDP form of communication is SOCK_DGRAM. The name of protocol is ignored for UDP and its value is set to zero. The socket() function returns a socket descriptor when it succeeds and the value INVALID_SOCKET when it fails. Client application must be able to locate and identify a server's socket, which is identified by a socket name that consists of IP address, port number, and protocol. Once the client socket has successfully contacted the server socket, the two names combine to form an association. This association establishes the identification of both sockets. Although UDP sockets are connectionless, most UDP applications use the same association for the life of the socket.

## C. BINDING SOCKET TO PORT

Setting the address family, local IP address and port number in the sockaddr_in structure initializes the socket. These three attributes form the socket name and they can be described as follows:

- The address family is the Internet address family PF_INET.
- Local IP address may be the value INADDR_ANY when requesting a local IP address to be assigned automatically.
- The port number identifies the network application protocol that other systems can send data to.

The attributes are assigned to a socket by the bind() function. This function takes in three arguments; socket handle, pointer to sockaddr_in, and length of this socket structure.  The bind() function returns zero if successful, and SOCKET_ERROR when it fails.

## D. SENDING A DATAGRAM

The sendto() function is used to send a datagram in UDP.  This function takes in six arguments namely; the socket handle, pointer to a buffer with outgoing data, length of data (in bytes) to send, flag to affect behavior of send function, pointer to destination socket structure (contains destination address and port number), and length of destination socket structure.  The flag is normally set to zero for general UDP transmission.

## E. RECEIVING A DATAGRAM

A datagram can be received by using the function, recvfrom().  This function takes in six arguments which are similar to those used in the sendto() function.  These arguments are namely; the socket handle, pointer to a buffer to receive data, length of data (in bytes) to receive, flag to affect the behavior of this function, pointer to source socket structure (contains source address and port number), and length of source socket structure.  The flag is normally set to zero for general UDP reception.

## F. CLOSING THE SOCKET

The closesocket() function is used to close the local socket connection for the system, and it takes in the specific socket handle to be closed as an argument.  Whenever this function is invoked, it returns the local socket resources to the protocol stack immediately.  The other systems that this system communicates with will not know that it has terminated its socket and stopped its communication.

**THIS PAGE INTENTIONALLY LEFT BLANK**

# IV. IMPLEMENTATION

## A. INTRODUCTION

This thesis implemented a three-dimensional virtual environment. It was created using the OpenGL software library and is coded in C++. Up to six PC workstations, each controlling an individual aircraft can interact with one another within the same virtual environment via Internet using either HLA or UDP protocol. The goal is to demonstrate, measure, and compare the performance of both HLA and UDP while executing identical virtual environment and simulation models. This implementation comprises five major components specifically, the terrain module, environment module, entity simulation modules, network modules, and performance measurement modules/tools. The following sections describe these modules in detail.

## B. TERRAIN MODULE

This module models a flat valley approximately 45 scale miles in diameter flanked by a ridge of mountains. Since all the entities reside in the same virtual world, this terrain remains unchanged throughout the simulation. Therefore, this terrain module is individually modeled by each participating workstation.

## C. ENVIRONMENT MODULE

A virtual sky dome of about 50 scale miles in height encloses the entire virtual terrain model. Each user is able to independently select eight different environments namely, dawn, day, dusk, or night (all of which may include or exclude the effects of fog). This module changes the hue of the virtual sky dome, in addition to the location and intensity of the light source (i.e., either sun or moon) to match the environment.

## D. ENTITY SIMULATION MODULES

This module contains the aircraft aerodynamic, missile aerodynamic, and floater models. Since all entity simulation models reside within each workstation, only changes to the entities are transmitted over the network. This reduces the overall network bandwidth requirement. Each workstation then computes its entities behavior locally, based either on the latest remote workstation updates or local commands received. Moreover, with this method of updating entities, responses can be implemented in either HLA or UDP form of data transmission without significant modification to the simulation software. The following sections describe each model in detail.

### 1. Aircraft Aerodynamic Model

The aerodynamic model of the aircraft resides in the simulation software, and as a result each participating workstation is only required to transmit its own aircraft commands e.g., roll left, roll right, climb, dive, increase thrust, and decrease thrust. A unique aircraft identification number is tagged to every command transmitted on the network. Every workstation will receive this command and simulate the effect of it on the specific aircraft.

Occasionally, some commands may not be received by the workstation and an aircraft may appear to have ‘drifted’ from its original position. In order to narrow this disparity, each participating workstation will periodically provide updates on its aircraft’s position (i.e., x, y, z), orientation (i.e., roll, pitch, heading), speed, and missile quantity.

### 2. Missile Aerodynamic Model

The simulation of this model is handled by the workstation that fires the missile, and only the result of the missile simulation (i.e., destroyed floater) is transmitted to the other workstations. The missile is used to target the floaters and not the other participating aircraft, thus firing a missile at the other aircraft will not destroy it. The missile aerodynamic model tracks the number of available missiles on the aircraft. It uses the aircraft's position and orientation at the point of missile fire to compute the missile trajectory. This missile trajectory takes into account the gravitation force exerted on the missile during its flight, as well as its fuel burnout time. The model has a self-destruct timer that will then destroy the missile after it has exceeded its fuel burnout time.

A missile top-up station is also modeled to replenish the number of missiles in each aircraft that flies through it.

### 3. Floater Model

This is the key model used to determine the performance of both HLA and UDP. Varying the number of floaters in the virtual environment varies the number of entities transmitted over the network. The floater follows a simple 'flight' path by circling in the virtual environment.

The unique floater identification number, status (i.e., destroyed or alive), and position (i.e., x, y, z) of each floater are continuously transmitted to the other workstations. Only one workstation will provide the floater updates to the other workstations. This privilege is allocated to the workstation with the largest aircraft identification number.

## E. NETWORK MODULES

The network modules consist of HLA and UDP forms of data transport. The following sections will cover the implementation of each module in detail.

### 1. High Level Architecture

This RTI implements an interaction class federate to transmit updates regularly on the network. This is similar to the UDP form of data transfer. Hence, we are able to closely match the performance of the two data transfer methods. The subsequent sub-sections will discuss the implementation of the HLA's RTI specifications.

#### *1.1 Federation Management - createFederationExecution()*

```
char* const fedExecName = "Fighter";                    // Name of the Federation Execution
try
{
        rtiAmb.createFederationExecution( fedExecName, "Fighter.fed" );
}
catch ( RTI::FederationExecutionAlreadyExists& e )
{
        cerr << "FED_HW: Note: Federation execution already exists." << &e << endl;
}
catch ( RTI::Exception& e )
{
        cerr << "FED_HW: ERROR:" << &e << endl;
}
```

The createFederationExecution method creates a new FedExec process and registers it with the RTI executive (RtiExec). After it has been registered, it will allow federates to join the new federation. This method takes in two arguments, the FedExec name to create, and the Federation Executive Data (FED) file name. The FED is a file found in the RTI configuration directory, and it defines the data classes and types that are exchanged in the federation.

The two exceptions that will be captured are Exception and FederationExecutionAlreadyExists. The latter exception is thrown when a FedExec for the given Federation has already been registered with the RtiExec. Upon termination of a FedExec, it has to unregister with the Rtiexec. An abnormal termination (e.g., kill –9) would result in an invalid FedExec that remains registered with Rtiexec. It must be manually unregistered at the Rtiexec console (i.e., remove <Federation Name>). The former exception is captured for all other exceptions.

### *1.2 Federation Management – joinFederationExecution()*

```
HwFederateAmbassador     fedAmb;
RTI::Boolean Joined = RTI::RTI_FALSE;
int numTries = 0;
while( !Joined && (numTries++ < 20) )
{
            try
            {
                        federateId = rtiAmb.joinFederationExecution( "FloaterOne",
                                             fedExecName,
                                              &fedAmb);
                        Joined = RTI::RTI_TRUE;
            }
            catch (RTI::FederateAlreadyExecutionMember& e)
            {
                        cerr << "FED_HW: ERROR: " << myHLA->GetName()
                                    << " already exists in the Federation Execution "
                                    << fedExecName << "." << endl;
                        cerr << &e << endl;
            }
            catch (RTI::FederationExecutionDoesNotExist&)
            {
                        cerr << "FED_HW: ERROR: " << fedExecName << " Federation Execution "
                                    << "does not exists."<< endl;
                        Sleep(2000);
            }
            catch ( RTI::Exception& e )
            {
                        cerr << "FED_HW: ERROR:" << &e << endl;
            }
}           // end of while
```

This method requests permission to participate in a federation execution and initializes the RTIambassador with federation specific data based on a FED file and the current execution status of the federation. Since the method createFederationExecution() does not synchronize the joining of federates to the newly created federation, the federate has to introduce a delay between the create and join events, or continuously invoke the join command until it succeeds. This method takes in three arguments namely, the name of the federate, the name of FedExec, and a pointer to the federateAmbassador.

The three exceptions captured are FederateAlreadyExecutionMember, FederationExecutionDoesNotExist, and Exception. The exception FederateAlready-ExecutionMember is thrown when the RTIambassador is already associated with a FedExec. The RTIambassador can only be associated with one FedExec at any given time. Nevertheless, it may be associated with other FedExec processes at a different time.

The exception FederationExecutionDoesNotExist is thrown when the RTI does not have the specified FedExec registered in the federation. All other exceptions are captured by Exception.

### *1.3 Federation Management – destroyFederationExecution()*

```
try
{
        rtiAmb.destroyFederationExecution( fedExecName );
}
catch ( RTI::Exception& e )
{
        cerr << "FED_HW: ERROR:" << &e << endl;
}
```

This method unregisters a federation and shut down the FedExec. The FedExec upon receiving this instruction would inform the RtiExec of its intention to tear down itself and then exit. Any federate can destroy the Federation by invoking this method. The federate need not be a member of the FedExec. All exceptions will be captured in the above implementation.

### *1.4 Federation Management – tick()*

```
try
```

```
{
        rtiAmb.tick();
}
catch ( RTI::Exception& e )
{
        cerr << "FED_HW: ERROR:" << &e << endl;
}
```

The zero argument tick method is implemented in this thesis to read all available network traffic, and then process the data as much as possible without blocking for additional network communications.

This method yields processor time from the federate to the LRC. This is **crucial** for RTI implementation because the LRC has to perform periodic federation maintenance like, sending federate heartbeats, updating interactions, and process incoming data from the network. Without this tick method, the RTI implementation will not function. All exceptions to this method are captured through Exception.

### *1.5 Time Management – enableTimeRegulation()*

```
RTIfedTime       grantTime(0.0);
const RTIfedTime timeStep(10.0);
try
{
        rtiAmb.enableTimeRegulation( grantTime, HLA::GetLookahead());
}
catch ( RTI::Exception& e )
{
        cerr << "FED_HW: ERROR:" << &e << endl;
}
```

This method instructs the federation to consider the federate's logical time for the purpose of governing the advancement of federation logical time. The grantTime is the minimum time to which the federate's logical time can be set when the time regulation mode is turned on. The GetLookahead method in the argument provides the length of the logical-time interval in addition to the federate's logical time at any given point in time. The sum of the federate's logical time and its lookahead is known as the effective logical time of the federate. This is the minimum permissible time-stamp for a time-stamp-ordered event generated by the federate. All exceptions are captured by Exception.

### *1.6 Time Management – enableTimeConstrained()*

```
try
{
		rtiAmb.enableTimeConstrained();
}
catch ( RTI::Exception& e )
{
		cerr << "FED_HW: ERROR:" << &e << endl;
}
```

This method instructs the LRC to deliver time-stamp-ordered events to the federate in an increasing order according to their associated time stamp. The time-stamp-ordered events will only be delivered to a time-constrained federate when a time-advancement service (i.e., timeAdvanceRequest()) is in progress. This event will not be delivered until the LRC guarantees that no events will be received with an earlier time-stamp. This guarantee is given when the federate receives a timeAdvanceGrant() callback service after it has made a timeAdvanceRequest(). All exceptions are captured by Exception.

### *1.7 Declaration Management – Interaction Publication and Subscription*

```
RTI::InteractionClassHandle   HLA::ms_planeTypeId              = 0;
RTI::ParameterHandle          HLA::ms_planeCommandTypeId       = 0;
RTI::ParameterHandle          HLA::ms_planeXTypeId             = 0;
RTI::ParameterHandle          HLA::ms_planeYTypeId             = 0;
RTI::ParameterHandle          HLA::ms_planeZTypeId             = 0;
RTI::ParameterHandle          HLA::ms_planeHeadingTypeId       = 0;
RTI::ParameterHandle          HLA::ms_planePitchTypeId         = 0;
RTI::ParameterHandle          HLA::ms_planeRollTypeId          = 0;
RTI::ParameterHandle          HLA::ms_planeSpeedTypeId         = 0;
RTI::ParameterHandle          HLA::ms_planeMissileTypeId       = 0;
RTI::ParameterHandle          HLA::ms_floaterXTypeId           = 0;
RTI::ParameterHandle          HLA::ms_floaterYTypeId           = 0;
RTI::ParameterHandle          HLA::ms_floaterZTypeId           = 0;
```

```
RTI::ParameterHandle        HLA::ms_floaterIdTypeId                 = 0;
RTI::ParameterHandle        HLA::ms_floaterDestroyedTypeId          = 0;

char* const  HLA::ms_planeTypeStr = "Plane";
char* const  HLA::ms_planeCommandTypeStr = "PlaneCommand";
char* const  HLA::ms_planeXTypeStr = "PlaneX";
char* const  HLA::ms_planeYTypeStr = "PlaneY";
char* const  HLA::ms_planeZTypeStr = "PlaneZ";
char* const  HLA::ms_planeHeadingTypeStr = "PlaneHeading";
char* const  HLA::ms_planePitchTypeStr = "PlanePitch";
char* const  HLA::ms_planeRollTypeStr = "PlaneRoll";
char* const  HLA::ms_planeSpeedTypeStr = "PlaneSpeed";
char* const  HLA::ms_planeMissileTypeStr = "PlaneMissile";
char* const  HLA::ms_floaterXTypeStr = "FloaterX";
char* const  HLA::ms_floaterYTypeStr = "FloaterY";
char* const  HLA::ms_floaterZTypeStr = "FloaterZ";
char* const  HLA::ms_floaterIdTypeStr = "FloaterId";
char* const  HLA::ms_floaterDestroyedTypeStr = "FloaterDestroyed";

ms_planeTypeId = ms_rtiAmb->getInteractionClassHandle( ms_planeTypeStr );
ms_planeCommandTypeId = ms_rtiAmb->getParameterHandle( ms_planeCommandTypeStr,
                                                        ms_planeTypeId);
ms_planeXTypeId = ms_rtiAmb->getParameterHandle( ms_planeXTypeStr, ms_planeTypeId);
ms_planeYTypeId = ms_rtiAmb->getParameterHandle( ms_planeYTypeStr, ms_planeTypeId);
ms_planeZTypeId = ms_rtiAmb->getParameterHandle( ms_planeZTypeStr, ms_planeTypeId);
ms_planeHeadingTypeId = ms_rtiAmb->getParameterHandle( ms_planeHeadingTypeStr,
                                                     ms_planeTypeId);
ms_planePitchTypeId = ms_rtiAmb->getParameterHandle( ms_planePitchTypeStr,
                                                     ms_planeTypeId);
ms_planeRollTypeId = ms_rtiAmb->getParameterHandle( ms_planeRollTypeStr, ms_planeTypeId);
ms_planeSpeedTypeId = ms_rtiAmb->getParameterHandle( ms_planeSpeedTypeStr,
                                                     ms_planeTypeId);
ms_planeMissileTypeId = ms_rtiAmb->getParameterHandle( ms_planeMissileTypeStr,
                                                     ms_planeTypeId);
ms_floaterXTypeId  = ms_rtiAmb->getParameterHandle( ms_floaterXTypeStr, ms_planeTypeId );
ms_floaterYTypeId  = ms_rtiAmb->getParameterHandle( ms_floaterYTypeStr,  ms_planeTypeId );
ms_floaterZTypeId  = ms_rtiAmb->getParameterHandle( ms_floaterZTypeStr, ms_planeTypeId );
ms_floaterIdTypeId  = ms_rtiAmb->getParameterHandle( ms_floaterIdTypeStr, ms_planeTypeId );
ms_floaterDestroyedTypeId    = ms_rtiAmb->getParameterHandle( ms_floaterDestroyedTypeStr,
                                                      ms_planeTypeId );

// Declare my Interaction interests
ms_rtiAmb->publishInteractionClass( ms_planeTypeId );
ms_rtiAmb->subscribeInteractionClass( ms_planeTypeId );
```

The getInteractionClassHandle() and getParameterHandle() methods convert the interaction class name and parameter name, respectively, to the RTI handles associated with them. These handles are used by the RTI services to refer to the interaction class and parameter. The getParameterHandle() requires the interaction class context of the parameter to be included as an argument.

The publishInteractionClass() method informs the LRC that the federate may begin producing interactions for a specific class. The federate will fail if it attempts to send the interactions of a class that it did not publish. The LRC will inform the publishing federate of the existence of remote subscribers.

The subscribeInteractionClass() method declares the federate's interest in receiving a specified class of interactions. The LRC will then deliver the interactions of the specified class to the federate when it is available on the network. Subscription to an interaction class involves subscription to all parameters within that class.

### *1.8 Object Management – Sending Interaction*

```
RTI::ParameterHandleValuePairSet* pParams = NULL;
pParams = RTI::ParameterSetFactory::create( 1 );

int flightCommand = flightInfo->command;
pParams->add( this->GetPlaneCommandRtiId(), (char*) &flightCommand, (sizeof(int)) );

try
{
        (void) ms_rtiAmb->sendInteraction( HLA::GetPlaneRtiId(), *pParams,
                                            this->GetLastTimePlusLookahead(),
                                            NULL );
}
catch ( RTI::Exception& e )
{
        cerr << "FED_HW: Error:" << &e << endl;
}
//------------------------------------------------------
// Must free the memory:
//    ParameterSetFactory::create() allocates memory on
//    the heap.
//------------------------------------------------------
delete pParams;
```

To send the parameters of an interaction class, the ParameterHandleValuePairSet is first initiated by allocating memory space using ParameterSetFactory::create(). The parameter is then packed into this allocated memory by invoking the add() method.

The sendInteraction() method takes in four arguments namely, the interaction class handle, the parameter value (packed in the ParameterHandleValuePairSet), the logical time used to determine the time-stamp-ordering of the interaction, and a tag containing information about the interaction (normally set to NULL). The allocated memory is freed after sending the parameter.

### *1.9 Object Management – Receiving Interaction*

```
void HLA::Update( RTI::InteractionClassHandle theInteraction,
        const RTI::ParameterHandleValuePairSet& theParameters )
{
        if ( theInteraction == HLA::GetPlaneRtiId() )
        {
                RTI::ParameterHandle paramHandle;
                RTI::ULong valueLength;
                for ( unsigned int j = 0; j < theParameters.size(); j++)
                {
                        paramHandle = theParameters.getHandle( j );
                        if ( paramHandle == HLA::GetPlaneCommandRtiId() )
                        {
                                int flight;
                                theParameters.getValue( i, (char*)&flight, valueLength );
                                m_command = flight;
                        }
                }
        }
}
```

To receive a parameter in an interaction class, the specific interaction class has to be identified by comparing it with its RtiId. Once the class is matched, the specific parameter within the interaction class needs to be identified.

The parameter handle is obtained by invoking the getHandle() method. Comparing the handle with the parameter RtiId identifies the specific parameter. The parameter is then extracted by the getValue() method, which obtains the specific parameter value from the ParameterHandleValuePairSet.

## 2. User Datagram Protocol

The UDP implementation for this thesis uses Windows sockets (WinSock) to establish lightweight connectionless data transport processes. This paradigm greatly reduces transport overhead as compared to the RTI implementation. For this implementation to work, it is necessary that the library module Wsock32.lib be present in the Link folder at Project Settings (hit Alt-F7 keys in Microsoft Visual C++ version 6.0). The following sub-sections will discuss the implementation of UDP transport service.

### *2.1 Establishing Contact*

```
WSADATA stWSAData;
SOCKET hSock;
bool ok = true;

WSAStartup(0x0101,&stWSAData);
hSock = socket(PF_INET, SOCK_DGRAM, 0);
if (hSock == INVALID_SOCKET)
{
        ok = false;
}
```

The method WSAStartup() must be the first WinSock function to be called. This is absolutely necessary in order to initialize the entire WinSock implementation. The first argument 0x0101 is the Winsock version number that the application requires. The code 0x0101 stands for WinSock version 1.01 (LSB is the major version, and MSB is the revision number). The second argument is a pointer to a buffer that the WinSock DLL will fill in.

The method socket() initializes the WinSock socket to be used for communication. The first argument PF_INET is a number that informs the socket that it is to be used for the Internet. The second argument SOCK_DGRAM informs the socket that the protocol to be used for communication is UDP. The last argument is the protocol name, this is usually set to zero. This method returns a socket handle when successful. This will be used for sending and receiving data. If it failed to establish a socket, it would return INVALID_SOCKET.

### *2.2 Binding Socket to Port*

```
SOCKADDR_IN stLclName;  /* local socket name (address & port) */
```

```
if (ok)
{
        stLclName.sin_family = PF_INET;
        stLclName.sin_port = htons(2882);
        stLclName.sin_addr.s_addr = INADDR_ANY;

        nRet = bind(hSock, (LPSOCKADDR) &stLclName, sizeof(struct sockaddr));
        if (nRet == SOCKET_ERROR)
        {
                ok = false;
        }
}
```

Before calling the method bind() to initialize the socket, the local socket name has to be established. The local socket name is comprised of the port and address numbers. The instruction, stLclName.sin_family = PF_INET, states that packets will travel on the Internet. The SOCKADDR data structure member, sin_port, establishes the local port 2882 that others can send data to. Finally, sin_addr.s_addr, uses the number INADDR_ANY to inform the socket that it is allowed to use any local IP address.

The method bind() then initializes the socket with the socket handle, a pointer to the local sockaddr_in data structure, and the length of this socket structure. This method returns zero, if successful. SOCKET_ERROR is returned when it fails.

### *2.3 Sending A Datagram*

```
SOCKADDR_IN stRmtName;  /* remote socket name (address & port) */
float dataOut[11];

stRmtName.sin_family = PF_INET;
stRmtName.sin_port = htons(2882);
stRmtName.sin_addr.s_addr = inet_addr("131.120.7.255");


dataOut[0] = (float) NUM_OF_FLOATERS;
dataOut[1] = (float) planeId;
dataOut[2] = flight.x;
dataOut[3] = flight.y;
dataOut[4] = flight.z;
dataOut[5] = flight.heading;
dataOut[6] = flight.pitch;
dataOut[7] = flight.roll;
dataOut[8] = flight.speed;
dataOut[9] = (float) flight.missile;
dataOut[10] = (float) flight.command;

sendto(hSock, (char FAR *)dataOut, sizeof(float)*11, 0,
                (LPSOCKADDR)&stRmtName, sizeof(SOCKADDR));
```

The method sendto() requires a remote socket name to send the data to. In the above code snippet, the sub-net IP address of the Graphics Lab was used to send the data, thereby broadcasting the data to all workstations belonging to that sub-net.

The data is transmitted by invoking the method sendto() by specifying in the arguments; socket handle, data array, size of the data, pointer to the socket structure, and the size of the socket structure.

### *2.4 Receiving A Datagram*

```
float dataIn[11];
int ret;
int nAddrSize = sizeof(SOCKADDR);

ret = recvfrom (hSock, (char FAR *)dataIn, sizeof(float)*11, 0,
                (struct sockaddr *)&stRmtName, &nAddrSize);

if (ret != SOCKET_ERROR)
{
        int firstValue = (int) dataIn[0];
        ident = (int)  dataIn[1];
        if (ident != planeId)
        {
                planeClass[ident]->setX(dataIn[2]);
                planeClass[ident]->setAltitude(dataIn[3]);
                planeClass[ident]->setZ(dataIn[4]);
                planeClass[ident]->setHeading(dataIn[5]);
                planeClass[ident]->setPitch(dataIn[6]);
                planeClass[ident]->setRoll(dataIn[7]);
                planeClass[ident]->setSpeed(dataIn[8]);
                missileClass[ident]->setMissile((int) dataIn[9]);
                planeClass[ident]->setCommand((int) dataIn[10]);
        }
}
```

The method recvfrom() obtains the data from the remote socket name. This method requires a valid socket handle, a pointer to a buffer   to store the data in, the number of bytes to send, optional flags which are set to zero, the remote socket name, and the size of the socket structure. The data is then extracted from the data array received from the remote socket name. This method returns the number of byes received when successful or SOCKET_ERROR on failure.

## F. PERFORMANCE MEASUREMENT MODULES/TOOLS

This module looks into the performance measurement of both HLA and UDP form of data transport. In order to determine the performance of these two protocols, four key measurement issues were identified. These are as follows, timeliness, network performance, graphics display performance, and CPU usage. Each of these performance issues is described in the following sub-sections.

### 1. Timeliness

Since the data packets received from HLA and UDP may not be in ordered sequence, timeliness of data is determined by measuring the number of data packets that arrived within a data set. For example, a block of data set may be an ordered sequence of packets numbered from zero to the maximum number of floaters (less one). When the data packets of the subsequent data block are received, it is assumed that data packets not received in the current data block are either lost or late. The timeliness of the current data block is then measured.

The rationale for implementing this method of measuring data timeliness is that late arriving data packets will be of no significant value to a real-time simulation system.

### 2. Network Performance

Network performance is measured by invoking ‘netstat –s 60’. This command provides periodic (60 second intervals) network performance statistics, such as number of datagrams received, number of errors received, and number of datagrams sent. This data provides useful network performance feedback for both HLA and UDP.

### 3. Graphics Display Performance

The average graphics display rate, in terms of frames per second, provides a truthful measurement of display realism in a real-time virtual environment. A frame rate of ten per second is the minimum rate acceptable to experience display realism. This display rate is obtained by measuring the time taken (in milliseconds) between frame displays by invoking the method timeGetTime() before and after rendering the frames and measuring the time difference.

### 4. CPU Usage

The CPU usage is measured on a per-task basis using a shareware tool, TaskInfo (http://www.iarsn.com/index.html#/download.html). This tool provides the average percentage CPU usage over a 60 second interval.

# V. RESULTS AND LIMITATIONS

## A. EXPERIMENT SETUP

In this experiment, three networked computers in the Graphics Laboratory were set up to execute identical virtual environment and simulation models using the HLA and UDP modes of data transport. To ensure that network traffic did not interfere with the results, all the experiments were conducted after office hours when the network traffic is relatively low.

The experiments consisted of three computers labeled Systems A, B and C. System C was responsible for updating all entity information. It was a primary sender of data, and System A was a primary receiver of this data. System B performed a role similar to A, however if B failed to receive the entity information from C it automatically assumed the role of a primary sender. In the HLA experiment setup, System A executed the rtiexec and fedexec processes simultaneously with the simulation application. Since System A was sending data from and receiving data for the rtiexec, in this setup it might be more appropriate to view System B as the primary receiver of the entity data. By determining the primary sender and receiver in the experiments, we could then approximate the data lost in both HLA and UDP modes.

The only variable factor in all the experiments was the number of entities. The performance measurements were taken as the number of entities increased from 10 to 3000. Between 10 and 350 entities, the number of entities was increased in step of 10. From 500 to 3000 entities, the step size was changed to 500.

The following performance measurements were obtained:

- Frame Rate. This is the number of frames displayed on the screen per second. Any frame rate of less than 10 frames per second would make the entities appear jerky on the display. The user would perceive that the entities were not getting updated smoothly. This would also inevitably make the controlling of the user's own aircraft model difficult.
- Timeliness. This is the percentage of the number of datagrams *received* within a batch of datagrams. In both the HLA and UDP implementations, all the entity updates were sent out in batches. At the next available transmission, a subsequent batch of entity information was sent. Timeliness is the measurement of the number of datagrams received from each batch before the subsequent batch arrived. The underlying purpose for this measurement is to determine the percentage of updated information received within a batch. Any late datagram received would not be useful in a real-time simulation system.
- Datagrams Sent. This is the rate at which the number of datagrams was sent per minute. This measurement was taken to compare the transmission rate between HLA and UDP as the number of entities increased.
- Datagrams Received. This is the rate at which the number of datagrams was received per minute. This measurement was taken to compare the receiving rate between HLA and UDP as the number of entities increased.
- Datagrams Lost. This is the percent of the number of datagrams received against the number of datagrams sent. In the UDP implementation, it was

easy to approximate the datagrams lost because System C was the primary sender and System A was the primary receiver. Therefore, the datagrams lost at System A was the difference between the number of datagrams sent at System C and received by System A. However, in the HLA implementation approximating the datagrams lost was difficult. This was mainly because HLA transmitted control information between federates to coordinate the data flow and manage the federation. Therefore the approximation of data lost was complicated by the additional control information transmitted by the federates. It was assumed that as the number of entities increased, the amount of control information compared to the amount of entity information became insignificant. The number of datagrams lost was approximated by measuring the difference between number of datagrams sent by all the federates and the number of datagrams received by the primary receiver, System B.

- Datagrams Error. This is the rate of the number of datagrams received in error per minute.
- CPU Usage. This is the average CPU utilization rate obtained in each experiment after 20 minutes of execution.

## B. SYSTEM CONFIGURATIONS

All the computers were operating on Windows NT Operating Systems platform. There was no additional software executing besides that which was required for this experiment. The three computers are labeled A, B and C throughout this experiment.

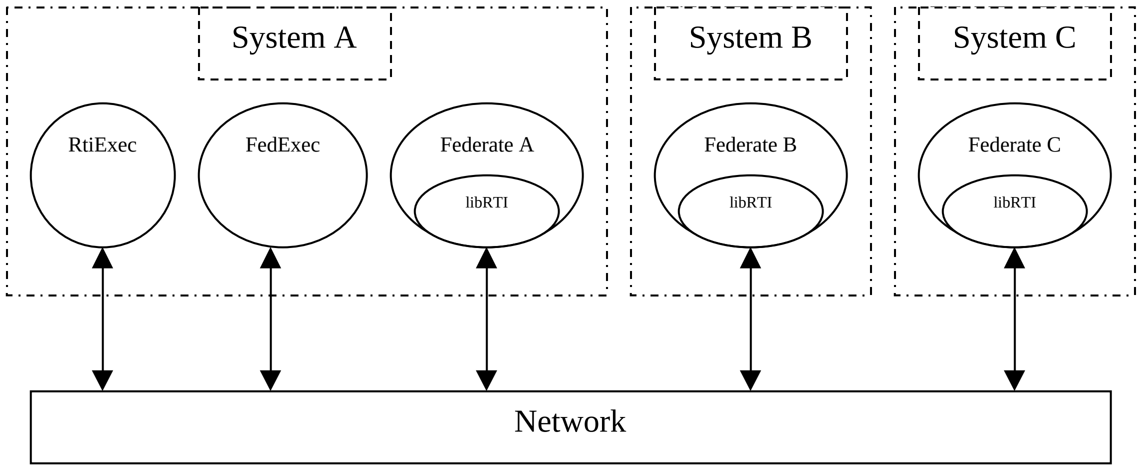

Figure 18. HLA Experiment Setup.

In HLA experiment setup (see Figure 18), System A executed the simulation application, the RtiExec, and the FedExec. As for System B and C, both systems executed only the simulation application.

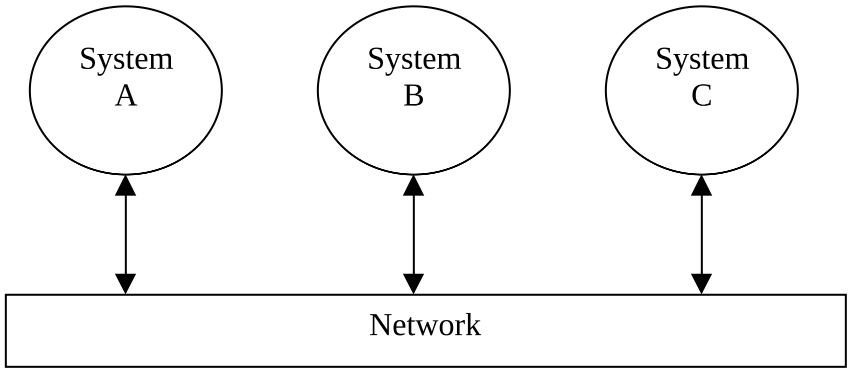

Figure 19. UDP Experiment Setup.

In the UDP experiment setup, all the systems were executing the simulation application. Their hardware configurations are as follows:

1. <u>System A</u>.

   CPU: x86 Family 6 Genuine Intel – 398 MHz

Memory: 128 MB

Video Card: ELSA Gloria

2. System B.

CPU: x86 Family 6 Genuine Intel – 398 MHz

Memory: 128 MB

Video Card: Real 3D Starfighter

3. System C.

CPU: Two x86 Family 6 Genuine Intel – 451 MHz

Memory: 256 MB

Video Card: Silicon Graphics Cobalt Graphics Chipset

System B had the best video card among the three computer systems. Although System C had two CPUs in its configuration, only one CPU was utilized during the experiment.

## C. RESULTS AND LIMITATIONS

The following sub-sections examine the experimental results.

### 1. Frame Rate

The number of frames per second for all three systems declined as the number of entities increased under both the HLA and UDP modes.

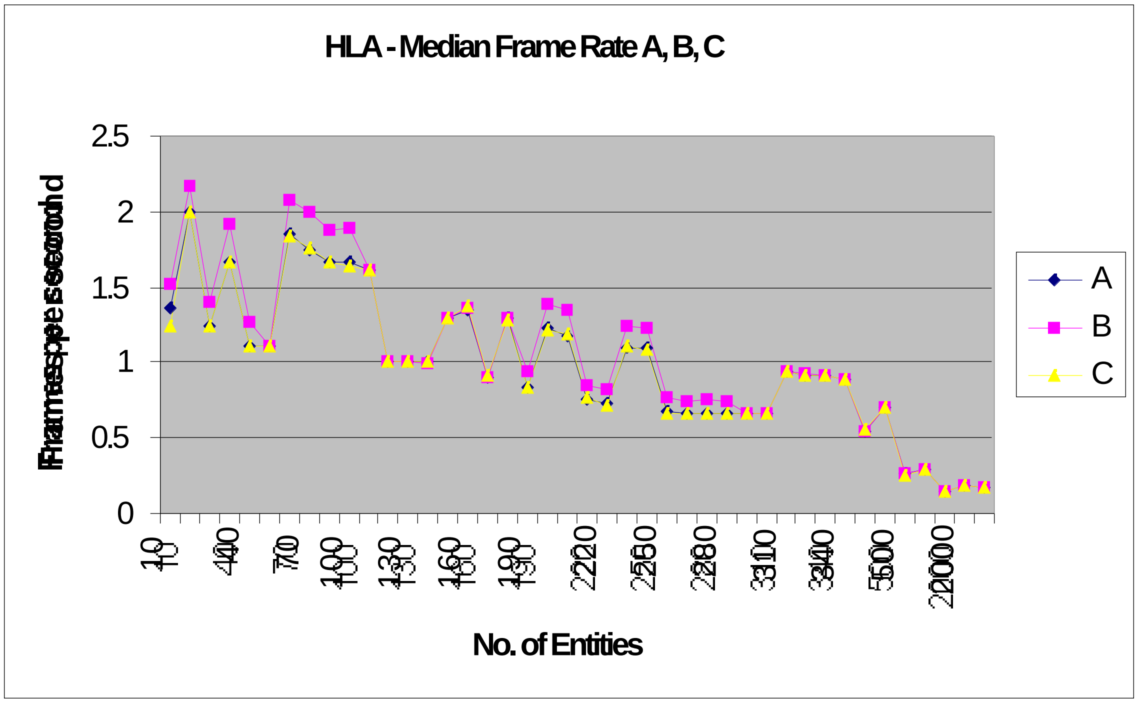

Figure 20. HLA – Median Frame Rate for Systems A, B, and C.

In Figure 20, although System B had the best graphics display card, there was only a slight improvement in its frame rate throughout all the experiments. It was observed that Systems A and C had similar performance even though System C had a much faster CPU clock rate and double memory capacity. The frame rate for all the systems appeared to have identical increase and dip characteristics.

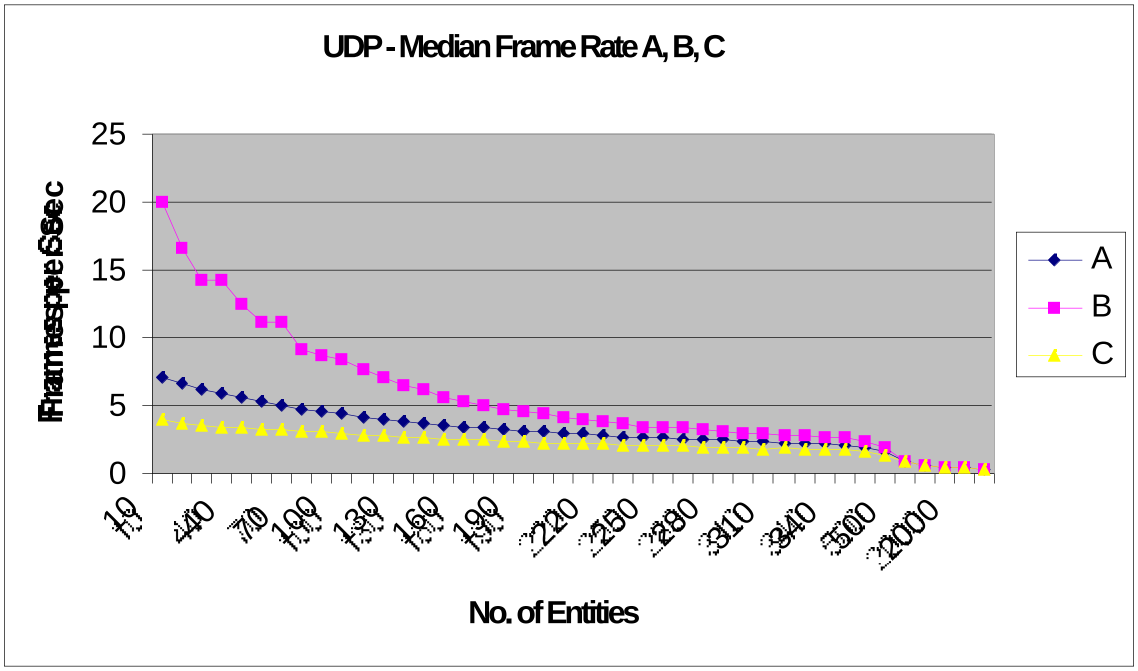

Figure 21. UDP – Median Frame Rate for Systems A, B, and C.

In Figure 21, System B showed a vast improvement in its frame rate as compared to the rest of the systems. Its performance then degraded to the same level as System A and C when the number of entities was approximately 1000. System A had a slightly better graphics card than System C therefore its frame rate was slightly better, albeit System C had better CPU and more memory.

As observed in all systems, UDP had allowed a higher frame rate than HLA by an order of magnitude. This is due to the fact that HLA required intensive utilization of the CPU. Most of these uses were dedicated to HLA execution. Since System B had the best video card, it performed remarkably well in UDP mode, when the CPU was not heavily utilized. Whereas in HLA mode, System B performed only moderately well.

## 2. Timeliness

Timeliness is measured only for System A, because throughout the experiment it was always receiving entity information from System C or B.

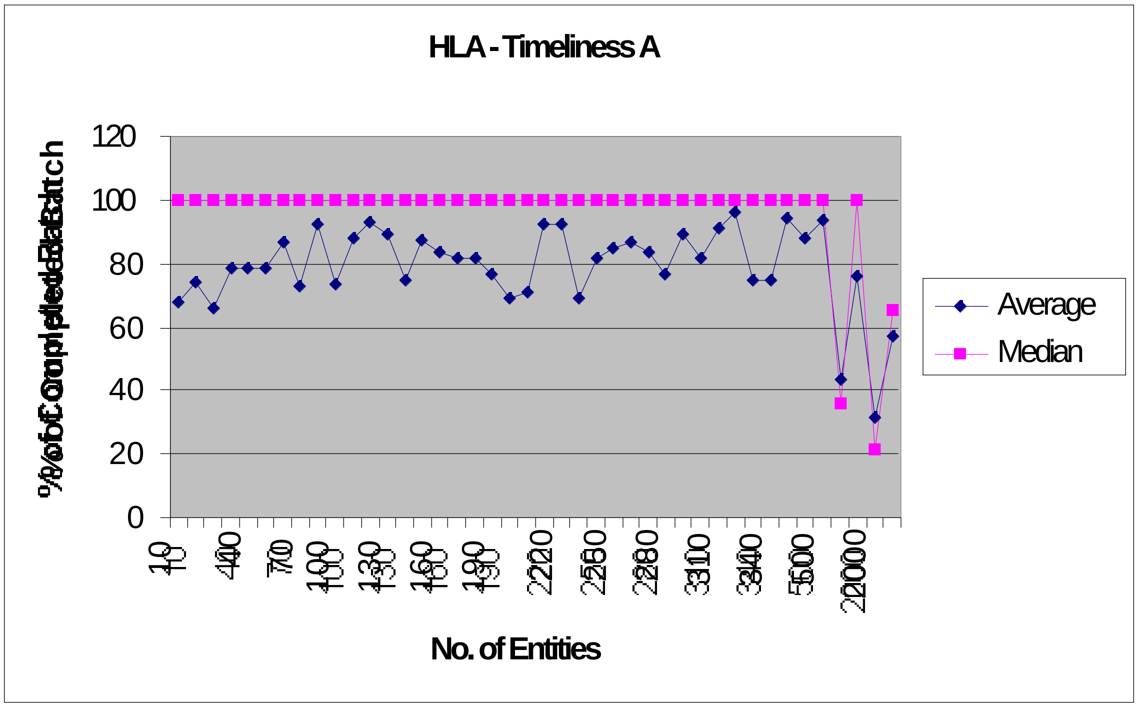

Figure 22. HLA – Timeliness for System A.

In Figure 22, most of the datagrams were received on time. The median line depicts this at 100%. On the average, HLA was able to maintain approximately 80% of the datagrams on time. The rest of the datagrams were either late or lost. As the number of entities approached approximately 1000, HLA performance started to degrade.

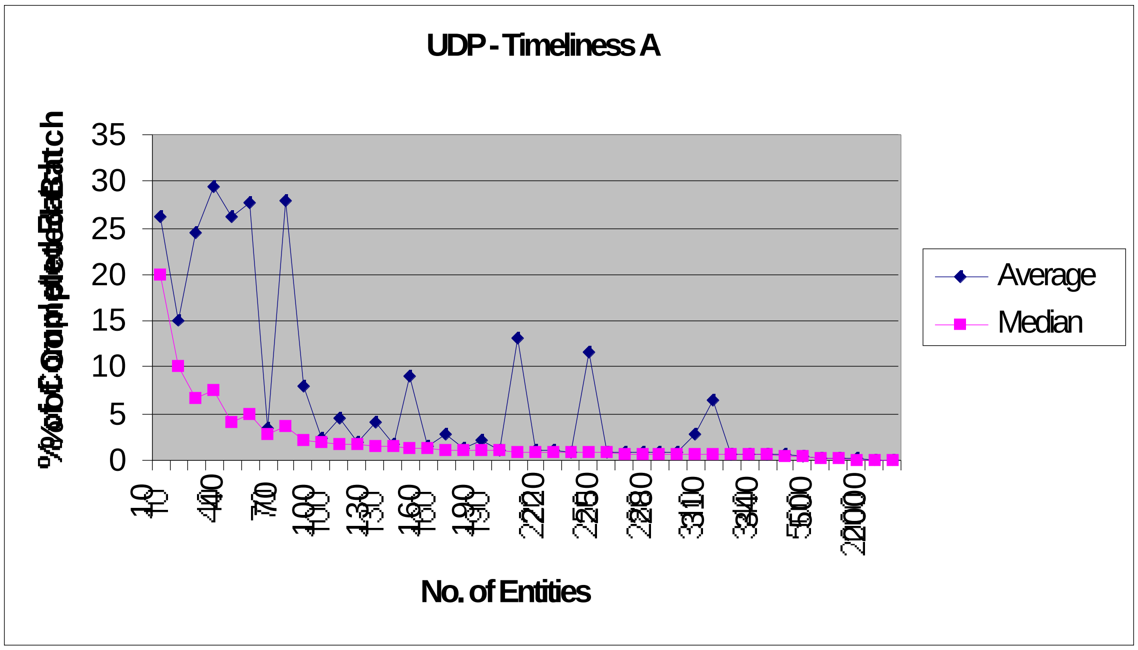

Figure 23. UDP – Timeliness for System A.

Figure 23 shows that the median timeliness of UDP performance decreased exponentially as the number of entities increased. When the number of entities approached approximately 100, less than 2% of the datagrams were received on time. The average timeliness performance fared slightly better however it was still undesirable.

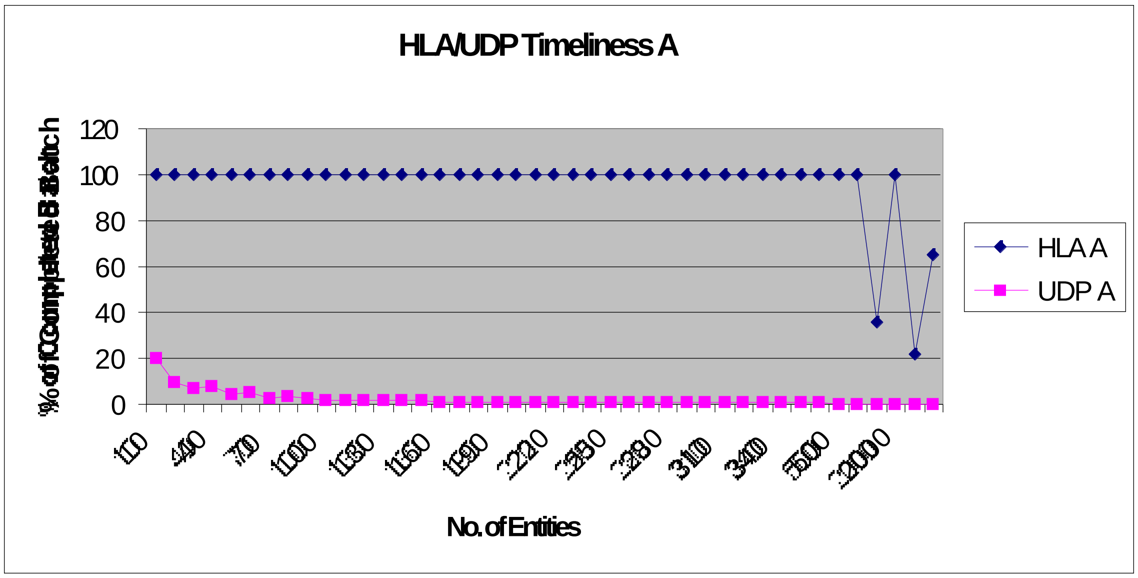

Figure 24. HLA/UDP – Timeliness for System A.

In Figure 24, the comparison between the two implementations shows that HLA performed extremely well with respect to timeliness and most of the packets in a batch were received prior to the arrival of the subsequent batch. However, in UDP mode the timeliness performance was exceptionally bad even for a small number of entities. This degradation in timeliness could be as a result of two factors, namely data losses and data not arriving on time.

### 3. Datagram Sent

The number of datagrams sent by the systems under both HLA and UDP mode is depicted below.

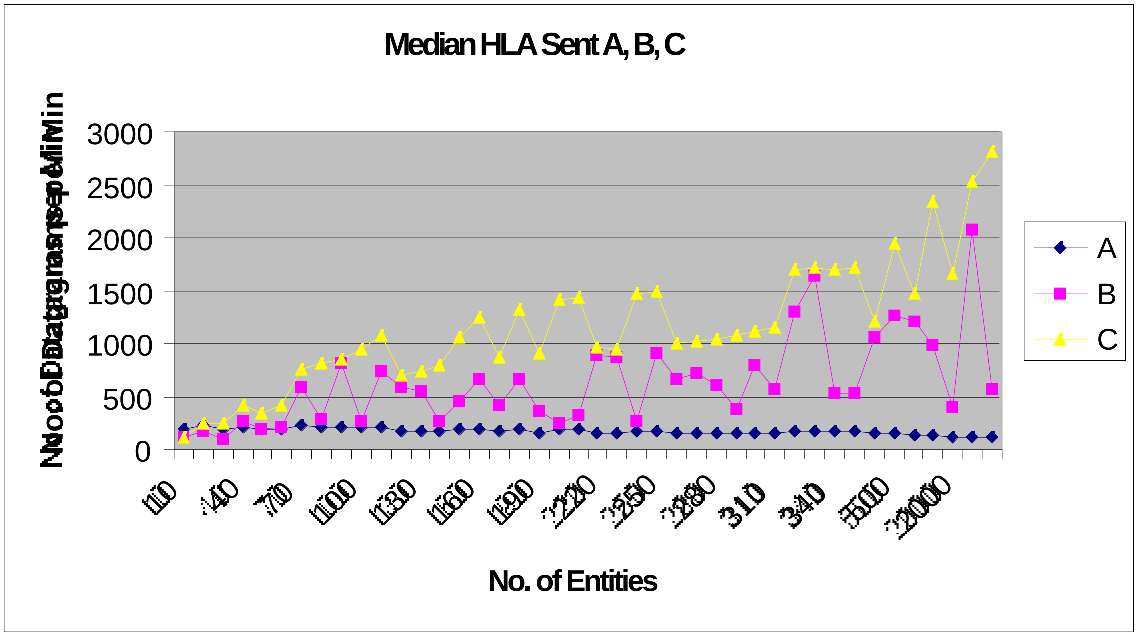

Figure 25. Median HLA Sent by Systems A, B, and C.

Figure 25 depicts all systems communicating with the rtiexec that resided in System A. System A in turn triggered the inter-process communication from the rtiexec to the federates. This kept the number of datagrams sent by System A consistently low throughout the experiment. The federates used the librti to implement methods which communicated with the rtiexec, fedexec and other federates. The FederateAmbassador identified the callback functions each federate was required to provide. These callback functions originated from the librti that resided in each federate. This allowed HLA to minimize data traffic by invoking callback functions instead of re-transmitting all the data received by the rtiexec.

Since System C was the predominant sender of the entity information, it had the highest number of datagrams sent per minute. System B sent some datagrams when it failed to receive the entity information from System C.

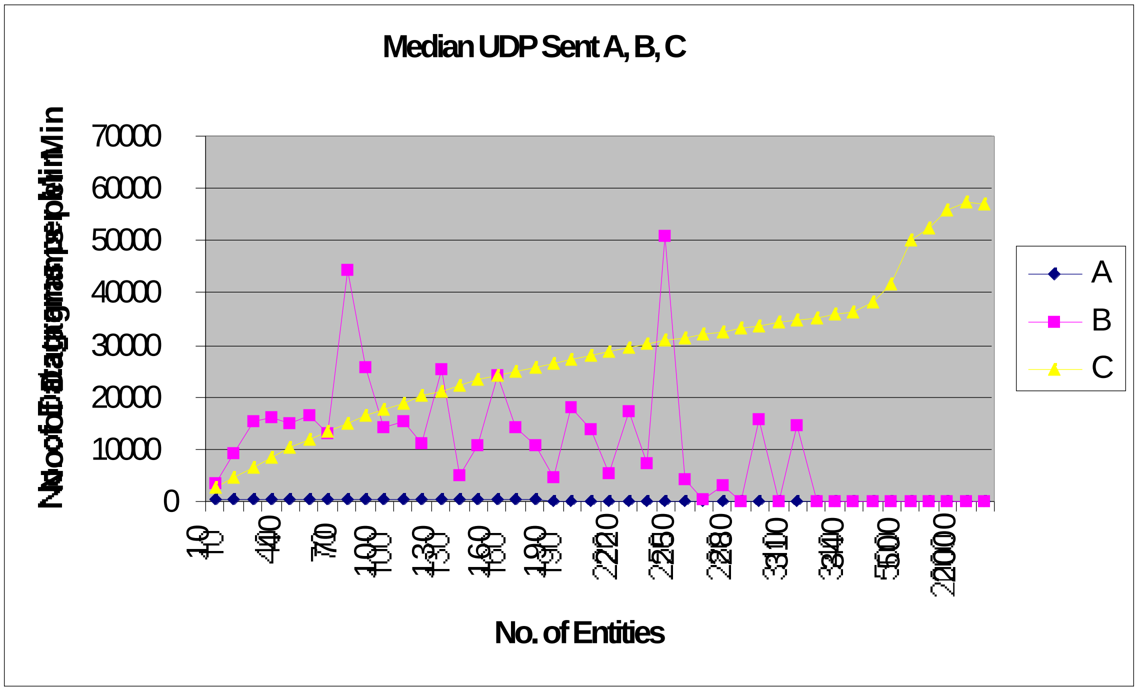

Figure 26. Median UDP Sent by Systems A, B, and C.

In Figure 26, System C was the predominant sender and hence it had the highest number of datagrams sent per minute. System B went into a send mode when it failed to receive the entity information from System C. Since System A was the predominant receiver of information and it did not send much information throughout the experiment.

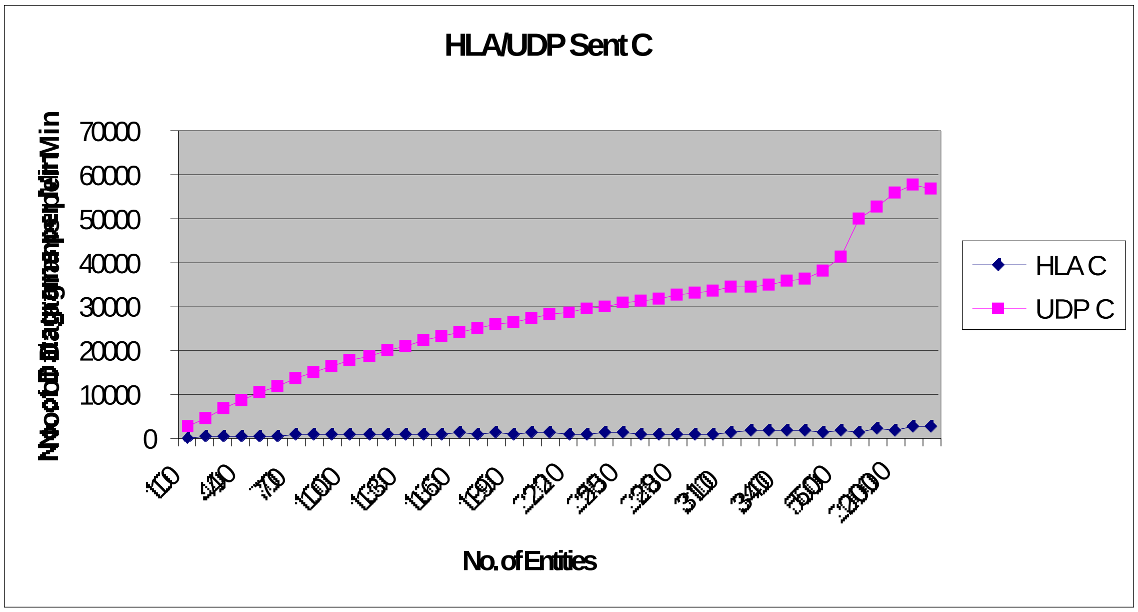

Figure 27. HLA/UDP Sent for System C.

It is observed in Figure 27 that System C, being the predominant sender of entity information, sent significantly less information while in HLA mode as compared to UDP mode. This indicates that HLA controlled the rate of data transmission during the session.

### 4. Datagrams Received

The number of datagrams received for both the HLA and UDP modes is depicted as follows.

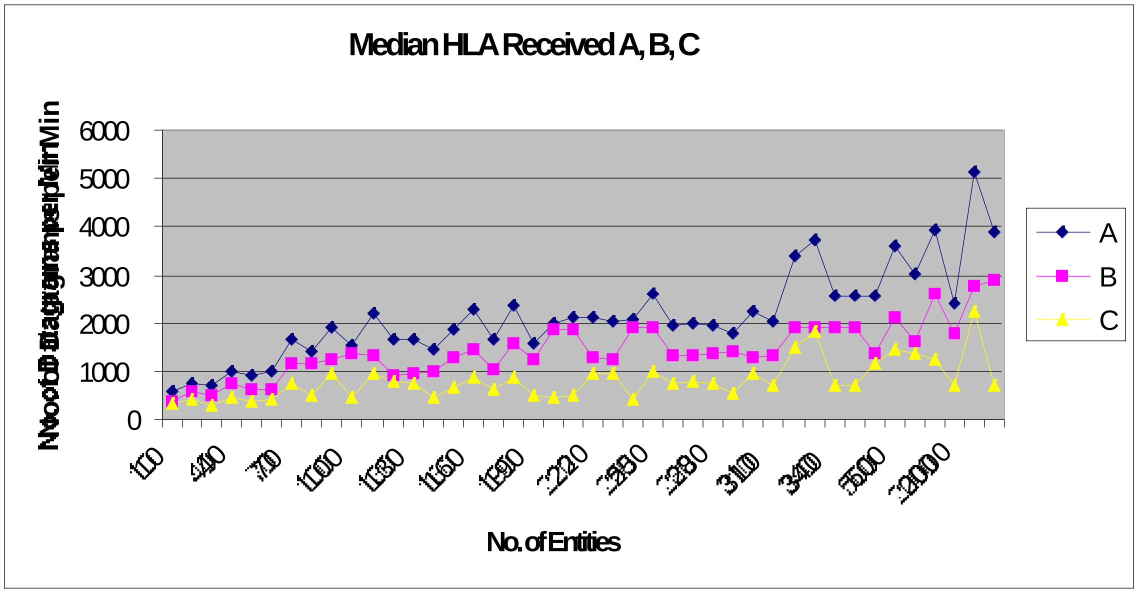

Figure 28. Median HLA Received for Systems A, B, and C.

In HLA mode (see Figure 28), System A received the highest number of datagrams per minute because the rtiexec resides in System A. Consequently, all federates in the experiment had to communicate with System A. In addition, Systems A and B were primarily the subscribers of the entity information from System C. Therefore, both systems showed a higher number of datagrams received compared to System C.

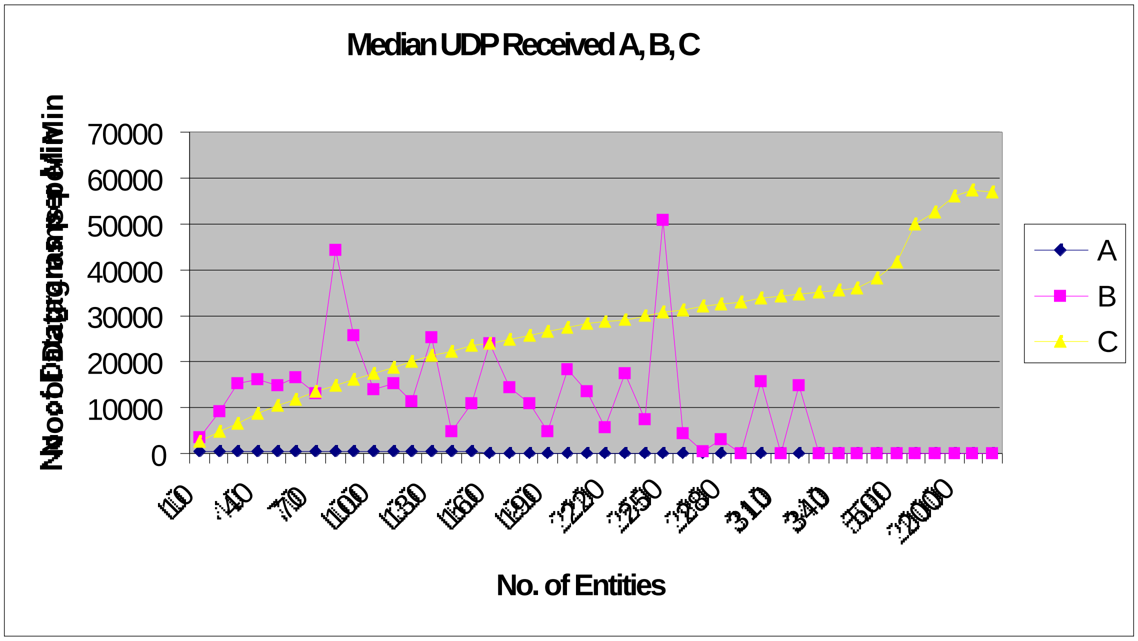

Figure 29. Median UDP Received for Systems A, B, and C.

However, in UDP mode (see Figure 29), System C received the most number of datagrams per minute. This is because System C was the only system to receive 100% of the datagrams that were sent by it. System B apparently received some datagrams because it only transmitted entity data when it failed to receive the data from System C, and therefore it received 100% of those datagrams.

## 5. Datagram Loss

Summing up the number of datagrams sent by all systems, and comparing that to the number of datagram received by System B approximates the number of datagrams lost under HLA. It would be appropriate to approximate the datagram loss per minute as the total number of datagrams received by System B per minute against the total number of datagrams sent by all systems.

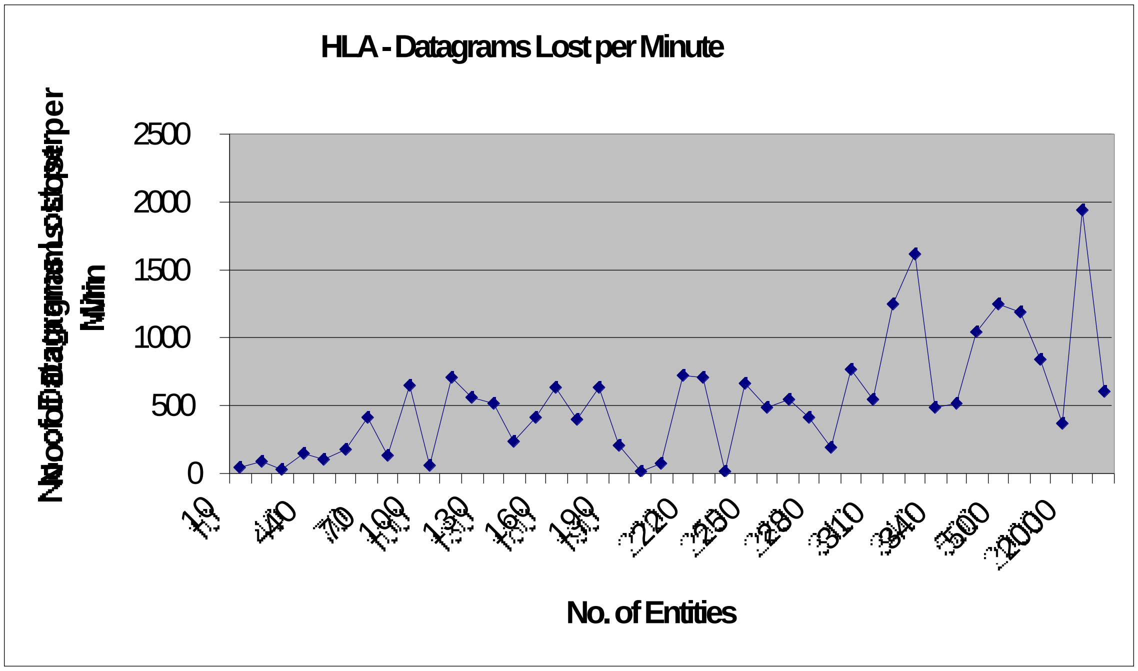

Figure 30. HLA Datagrams Lost per Minute.

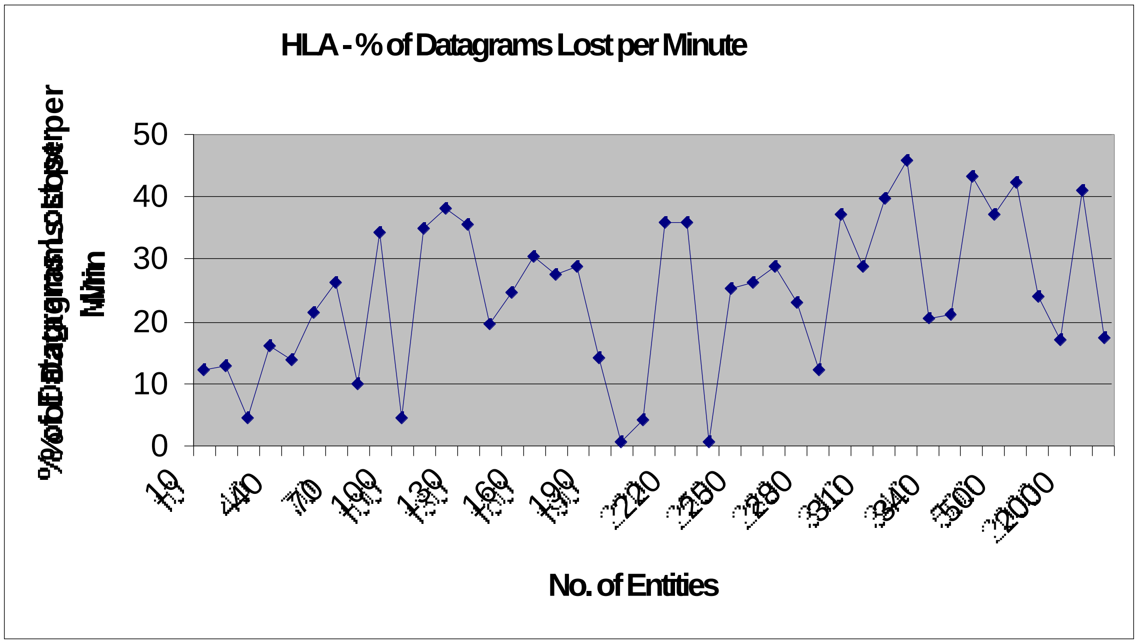

Figure 31. HLA - % of Datagrams Lost per Minute.

In Figure 31, it can be seen that approximately 30% of the datagrams transmitted in HLA mode was lost. This lost appeared to remain consistent in the 15% to 45% band as the number of entities increased.

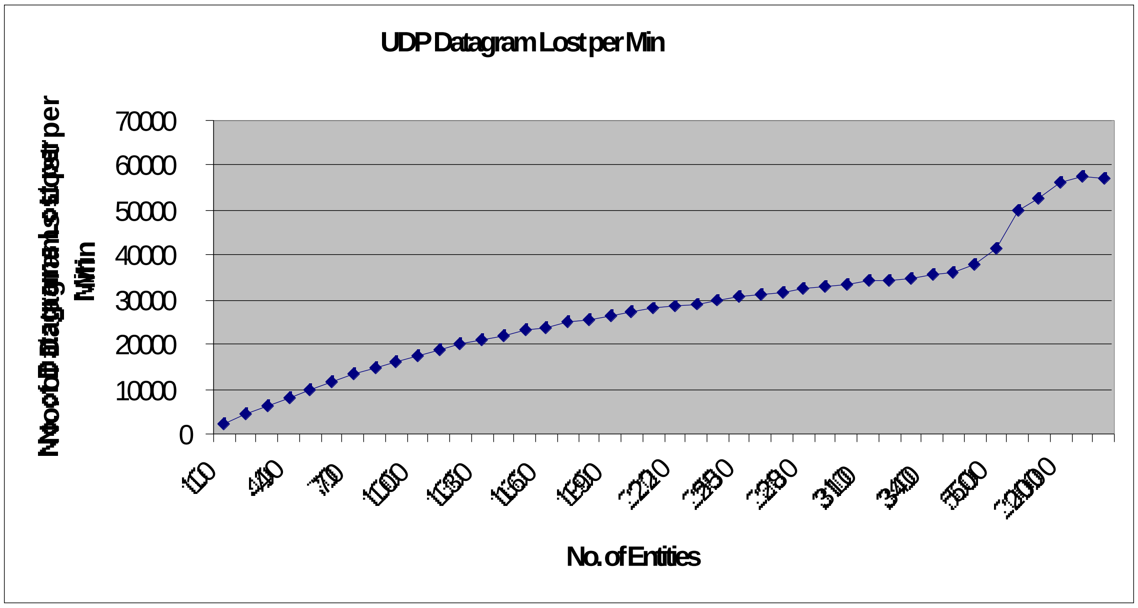

Figure 32. UDP Datagrams Lost per Minute.

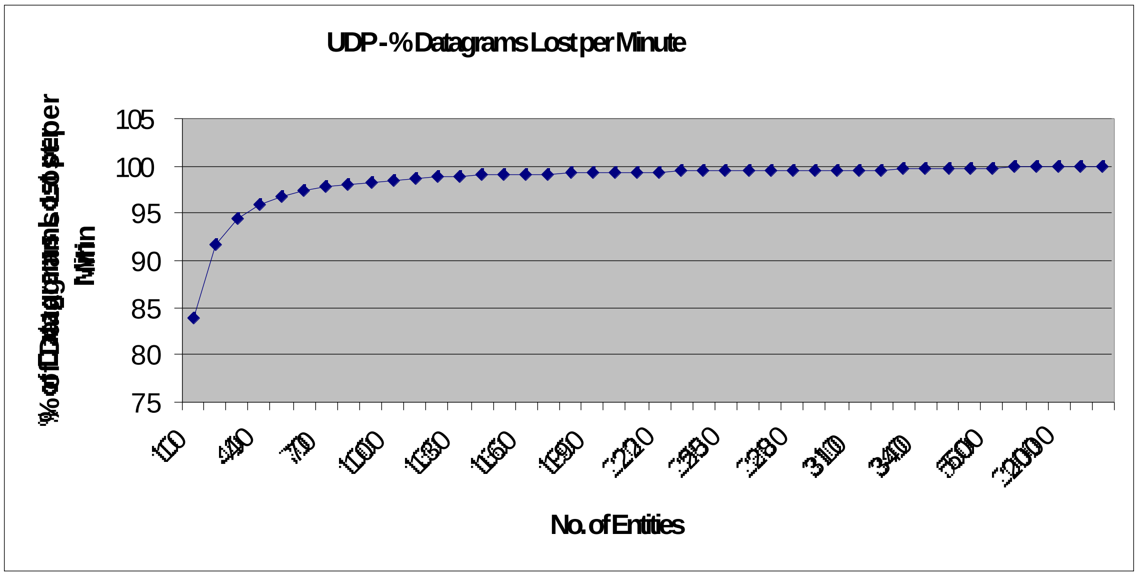

Figure 33. UDP - % of Datagrams Lost per Minute.

Figure 32 shows the number of datagrams lost per minute under UDP. Figure 33 depicts the percentage of datagrams lost per minute as the number of datagrams sent per minute increased. It is observed that UDP mode had extremely high datagram losses per minute that increased exponentially. This resulted in poor timeliness performance, since most of the datagrams were lost during transmission.

In a separate experiment, when the rate at which the datagrams were sent from System C was reduced, there was no significant change in the number of datagrams received by both Systems A and B. This might be because Systems A and B were receiving the entity information at a consistent rate. Both systems failed to receive the data when the receiving buffers became full and began to discard data from System C.

### 6. Datagrams Error

There was no datagram error observed throughout the experiments for either HLA or UDP.

### 7. CPU Usage

The CPU processors for System A, B and C were dedicated to this simulation. Each showed a 98% utilization rate. System C with the dual processors, utilized about 98% of 1 CPU while the other CPU was in an idle state during the experiment.

# VI. CONCLUSIONS

## A. SUMMARY

HLA provides good timeliness and minimizes data losses because it reduces the data traffic by invoking the callback functions that reside in each federate and controlling the number of datagrams sent per minute to federates of the federation. However, it utilizes extensive CPU cycles, which results in fewer cycles for rendering and other tasks. Installing a better graphics card or adding more memory will not resolve this problem. Since HLA is single threaded, having multiple processors will not help.

UDP has poor timeliness and high data losses because it uses extensive network bandwidth. However the actual overhead and demand for CPU cycles is much less than HLA. Therefore, more cycles can be used for rendering and other tasks.

## B. CONCLUSION

HLA is an improvement over its predecessor in terms of timeliness and data losses. These qualities could lead to more consistent and cohesive networked simulation. However the high overhead is a problem, as too few cycles are allocated for other tasks unless high MIPs computers such as the SGI Octane or Onyx class computers are used. Since HLA is single threaded a multi processor system cannot be used to resolve the high overheads. HLA should be multi-threaded if we wanted to take advantage of its good qualities by using multi-processor systems.

The RTI is too cumbersome to be handled by PC workstations for any real time simulation. Nevertheless, it could be employed as war-gaming simulation systems that do not require any real time response.

## C. FUTURE WORK

The following lists future projects that could provide greater insight into HLA and hence, improve its performance during implementation.

### 1. Reliable HLA And TCP Performance Measurement

This work looks into the performance comparison between reliable data transfer under HLA and TCP mode of data transport. Since reliable data under HLA uses TCP as the primary mode of data transport, the comparison between these two forms of data transport would provide details about how HLA performance could be improved during implementation.

### 2. Interaction and Object HLA Performance Measurement

HLA provides two classes of Object Model Template, that is, interactions and objects. A federation can be described completely in term of interactions, objects or both. This work requires the implementation of simulation systems that use solely interactions, and solely objects. Both classes would be compared to determine the performance effects of either system.

# LIST OF REFERENCES

1. Dahmann Judith, Kuhl Frederick, Weatherly Richard, *Creating Computer Simulation Systems – An Introduction to the High Level Architecture*, Prentice Hall, 1999.

2. Zyda Michael, Singhal Sandeep, *Networked Virtual Environments – Design and Implementation*, Addison Wesley, 1999.

3. Quinn Bob, Shute Dave, *Windows Sockets Network Programming*, Addison Wesley, 1999.

4. "*High Level Architecture – Run-Time Infrastructure Programmer's Guide 1.3v6*", Defense Modeling and Simulation Office, 12 March 1999, http://hla.dmso.mil

5. "*High Level Architecture – Run-Time Infrastructure (HLA-RTI) 1.3v6*", Defense Modeling and Simulation Office, 12 March 1999, http://hla.dmso.mil

THIS PAGE INTENTIONALLY LEFT BLANK

## INITIAL DISTRIBUTION LIST